Simulation Native des Systèmes Multiprocesseurs sur Puce à l'aide de la Virtualisation Assistée par le Matériel

Mian Muhammad Hamayun

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Mian Muhammad HAMAYUN

Thèse dirigée par **Frédéric PÉTROT**

préparée au sein du **Laboratoire TIMA**
et de l'École Doctorale **Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Simulation Native des Systèmes Multiprocesseurs sur Puce à l'aide de la Virtualisation Assistée par le Matériel

« Native Simulation of Multi-Processor System-on-Chip using Hardware-Assisted Virtualization »

Thèse soutenue publiquement le **04 juillet, 2013**,
devant le jury composé de :

Mme. Florence MARANINCHI
Professeur, Institut Polytechnique de Grenoble, Présidente
M. Jean-luc DEKEYSER
Professeur, Université des Sciences et Technologies de Lille, Rapporteur
M. Guy BOIS
Professeur Titulaire, École Polytechnique de Montréal, Rapporteur
M. Alain GREINER
Professeur, Université Pierre et Marie Curie (Paris VI), Examinateur
M. Benoit DUPONT DE DINECHIN
Directeur du Développement Logiciel, KALRAY, Examinateur
M. Frédéric PÉTROT
Professeur, Institut Polytechnique de Grenoble, Directeur de Thèse

« I dedicate this dissertation to my father, *Ashraf Ali Shad*, whose love and affection has no bounds and his lifelong efforts helped me reach where I stand today »

Abstract

Integration of multiple heterogeneous processors into a single *System-on-Chip* (**SoC**) is a clear trend in embedded systems. Designing and verifying these systems require high-speed and easy-to-build simulation platforms. Among the software simulation approaches, native simulation is a good candidate since the embedded software is executed natively on the host machine, resulting in high speed simulations and without requiring instruction set simulator development effort. However, existing native simulation techniques execute the simulated software in memory space shared between the modeled hardware and the host operating system. This results in many problems, including address space conflicts and overlaps as well as the use of host machine addresses instead of the target hardware platform ones. This makes it practically impossible to natively simulate legacy code running on the target platform. To overcome these issues, we propose the addition of a transparent address space translation layer to separate the target address space from that of the host simulator. We exploit the *Hardware-Assisted Virtualization* (**HAV**) technology for this purpose, which is now readily available on almost all general purpose processors. Experiments show that this solution does not degrade the native simulation speed, while keeping the ability to accomplish software performance evaluation. The proposed solution is scalable as well as flexible and we provide necessary evidence to support our claims with multiprocessor and hybrid simulation solutions. We also address the simulation of cross-compiled *Very Long Instruction Word* (**VLIW**) executables, using a *Static Binary Translation* (**SBT**) technique to generate native code that does not require run-time translation or interpretation support. This approach is interesting in situations where either the source code is not available or the target platform is not supported by any retargetable compilation framework, which is usually the case for VLIW processors. The generated simulators execute on top of our HAV based platform and model the *Texas Instruments* (**TI**) C6x series processors. Simulation results for VLIW binaries show a speed-up of around two orders of magnitude compared to the cycle accurate simulators.

Key Words

Design Automation, Simulation, System Level Design, *Hardware-Assisted Virtualization* (**HAV**), *System-on-Chip* (**SoC**), *Very Long Instruction Word* (**VLIW**), *Static Binary Translation* (**SBT**).

Résumé

L'intégration de plusieurs processeurs hétérogènes en un seul système sur puce (SoC) est une tendance claire dans les systèmes embarqués. La conception et la vérification de ces systèmes nécessitent des plateformes rapides de simulation, et faciles à construire. Parmi les approches de simulation de logiciels, la simulation native est un bon candidat grâce à l'exécution native de logiciel embarqué sur la machine hôte, ce qui permet des simulations à haute vitesse, sans nécessiter le développement de simulateurs d'instructions. Toutefois, les techniques de simulation natives existantes exécutent le logiciel de simulation dans l'espace de mémoire partagée entre le matériel modélisé et le système d'exploitation hôte. Il en résulte de nombreux problèmes, par exemple les conflits l'espace d'adressage et les chevauchements de mémoire ainsi que l'utilisation des adresses de la machine hôte plutôt des celles des plates-formes matérielles cibles. Cela rend pratiquement impossible la simulation native du code existant fonctionnant sur la plate-forme cible. Pour surmonter ces problèmes, nous proposons l'ajout d'une couche transparente de traduction de l'espace adressage pour séparer l'espace d'adresse cible de celui du simulateur de hôte. Nous exploitons la technologie de virtualisation assistée par matériel (HAV pour Hardware-Assisted Virtualization) à cet effet. Cette technologie est maintenant disponibles sur plupart de processeurs grande public à usage général. Les expériences montrent que cette solution ne dégrade pas la vitesse de simulation native, tout en gardant la possibilité de réaliser l'évaluation des performances du logiciel simulé. La solution proposée est évolutive et flexible et nous fournit les preuves nécessaires pour appuyer nos revendications avec des solutions de simulation multiprocesseurs et hybrides. Nous abordons également la simulation d'exécutables cross-compilés pour les processeurs VLIW (Very Long Instruction Word) en utilisant une technique de traduction binaire statique (SBT) pour généré le code natif. Ainsi il n'est pas nécessaire de faire de traduction à la volée ou d'interprétation des instructions. Cette approche est intéressante dans les situations où le code source n'est pas disponible ou que la plate-forme cible n'est pas supportée par les compilateurs reciblable, ce qui est généralement le cas pour les processeurs VLIW. Les simulateurs générés s'exécutent au-dessus de notre plate-forme basée sur le HAV et modélisent les processeurs de la série C6x de Texas Instruments (TI). Les résultats de simulation des binaires pour VLIW montrent une accélération de deux ordres de grandeur par rapport aux simulateurs précis au cycle près.

Mots Clès

Conception Assistée par Ordinateur (CAO), Simulation, Conception Niveau Système, Virtualisation Assistée par le Matériel (HAV), Système sur Puce (SoC), Mot d'Instruction Très Long (VLIW), Traduction Binaire Statique (SBT).

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Frédéric Pétrot for giving me the opportunity to work in his research group and for his continuous support during the last four years. I am also thankful to him for his guidance, kindness, patience and his technical support during research and writing of this dissertation. Without his help and guidance, it would have been impossible for me to finish this thesis.

I would like to thank Prof. Florence Maraninchi for presiding over my thesis defense committee. My special thanks to Prof. Jean-luc Dekeyser and Prof. Guy Bois for taking the time to review my thesis manuscript and their constructive remarks. I am very thankful to the examiners, Prof. Alain Greiner and Dr. Benoit Dupont De Dinechin for their pertinent questions and valuable remarks.

I would like to thank the French Ministry of Higher Education and Research for providing the financial support during my PhD studies. I am also thankful to Higher Education Commission (HEC) Pakistan, for awarding me the scholarship for Master program in France, which lead to my PhD studies.

I would like to thank my colleagues, Patrice Gerin, Hao Shen and Marius Gligor[1] for their indispensable support during my thesis. I must also thank the permanent SLS team members for their warm welcome and wonderful company, including Prof. Frédéric Rousseau, Hamed Sheibanyrad, Stephane Mancini, Paul Amblard (Late), Nicolas Fournel and Olivier Muller[2]. I should not forget Xavier Guerin, Luc Michel[1], Ashraf Elantably[1], Maryam Bahmani[1], Sahar Foroutan, Clément Deschamps, Etienne Ripert, Gilles Bizot, Wassim Mansour[1], Yi Gang[1] and Saif-ur-Rehman[1] for their support. I would like to convey my best wishes to Guillaume Sarrazin, who has accepted to continue working on native simulation technique for his thesis.

In my social circles, I would like to thank my dear friend Dr. Anis-ur-Rehman Khan, who provided me the best possible fellowship one could expect from anyone. His presence of mind, witty humor and sporting activities provided me with the necessary fresh air, during stressful times. I must not forget Asif Iqbal Baba for his religious expressions and cricketing enthusiasm, which brought the necessary physical activity to our lives. I am also very grateful to the Grenoblois Pakistanais community, which organized many gatherings that gave us the means to celebrate our cultural events, with friends and families.

In the end, I must thank my parents, my brothers Usman & Umar, my sister Ayesha, my wife Fouzia and my children Shehryar & Mahnoor for their prayers and encouragement during my studies. Without their moral support, things would have been much more difficult. I will remain in their debt for the rest of my life!

Mian Muhammad Hamayun
25 September, 2013

[1] An additional thanks for their help in the preparations of my "Pot-de-Thèse".

[2] An additional thanks for his kind support in my teaching assignments as well as in general discussions.

List of Figures

Page

List of Tables

Contents

Part I

Résumé Français

Le temps est un grand maître, dit-on, le malheur est qu'il tue ses élèves.

Berlioz

1 Simulation Native de MPSoC à l'aide de la Virtualisation Assistée par le Matériel

L'état actuel de l'art en matière de technologie de VLSI (VLSI pour Very-Large-Scale Integration) permet la fabrication de plusieurs milliards de transistors sur une seule puce. La conception et l'intégration du matériel avec un si grand nombre de transistors est très difficile. Une solution simple pour les concepteurs de matériel est de mettre plusieurs processeurs relativement simples sur une seule puce plutôt que de concevoir une machine mono-processeur complexe. Par ailleurs, l'évolutivité des mono-processeurs est limitée par les facteurs de dissipation de puissance, comme on ne peut pas simplement augmenter la vitesse d'horloge pour obtenir d'avantage de performance.

Dans le domaine des systèmes intégrés, des problèmes de consommation et de rendement nécessitent l'utilisation d'architectures simples. Il s'agit notamment de processeurs qui disposent un rapport MIPS (MIPS pour Million Instructions Per Second) plus elevé par rapport à la consommation en Watt. Cette tendance se poursuit vers des structures plus régulières et homogènes mais comprenant encore, éventuellement, du matériel spécifique et des extensions de jeu d'instructions dans les éléments de traitement. Ces multiprocesseurs spécifiques à une classe d'application sont appelés MPSoC (MPSoC pour Multi-Processor System-on-Chip). De nombreux secteurs industriels, tels que les télécommunications, l'aérospatiale, l'automobile, le militaire et l'électronique grand public utilisent actuellement ces systèmes.

1.1 Introduction

La complexité de conception de MPSoC est en augmentation à cause du nombre et du type d'éléments de calcul inclus dans ces systèmes. Selon l'ITRS, jusqu'à 6000 processeurs sont attendus sur un seul SoC d'ici la fin de l'année 2026, comme indiqué dans la Figure 1.1. La conception et la vérification d'une solution matérielle avec tel nombre d'éléments de calcul, qui de plus tourne un logiciel encore plus complexe, est un énorme défi.

De même, les délais de mise sur le marché sont d'une importance primordiale dans

l'électronique grand public, pour les fabricants de produits pour rester compétitif et de réussir à répondre aux exigences du marché. Les delais dans la conception et la fabrication de systèmes MPSoC plus en plus complexe minent cet objectif, car le développement du logiciel dépend de la disponibilité du matériel. Les plates-formes virtuelles, ou modèles de simulation, offrent une alternative, car le développement de logiciels peut commencer tôt et en parallèle avec la conception du matériel. Par ailleurs, les équipes de développement du logiciel ont besoin des plateformes de simulation, même si le matériel est disponible, car les premiers prototypes sont généralement très coûteux.

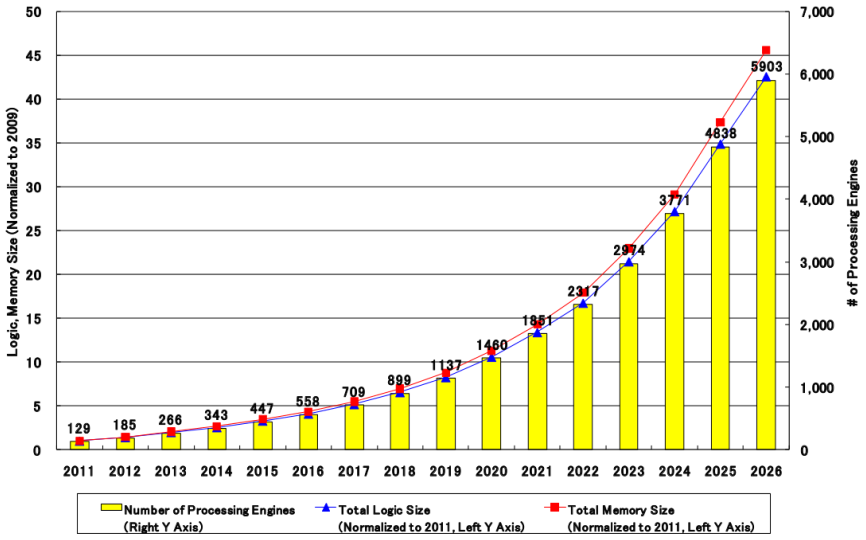


Figure 1.1: Les tendances de marché SoC et complexité de conception des systèmes portable [ITR]

La simulation s'appuie sur des modèles et comme il peut être entendu que si les modèles sont proches de la réalité, la simulation sera lente, et s'ils sont abstraits, la précision des résultats peut être contestable. Plusieurs strategies de simulation existent pour l'exécution du logiciel : la première s'appuie sur l'ISS (ISS pour Instruction Set Simulator) qui est la technologie de virtualisation du processeur la plus mature et couramment utilisée. L'ISS exécute les fichiers binaires cible (target) cross-compilé (Système d'Exploitation et Applications) et imitent le comportement des processeurs cibles par l'interprétation exacte des instructions [Bel73]. Leur principal inconvénient vient de la vitesse de simulation très lente. La seconde repose sur l'émulation, principalement par DBT [Bel05] (DBT pour Dynamic Binary Translation) mais nécessite des coûts de développement relativement élevé. La troisième utilise la simulation native, c'est-à-dire l'exécution de code sans compilation croisée. Des techniques de simulation natives [BBY+05, JBP06] atteingnent des vitesses de simulation supérieures comme ils surmontent l'interprétation / coût de la traduction, par la compilation du logiciel embarqué au format binaire hôte et l'exécutent directement sur les processeurs hôtes. Afin d'accéder à des ressources matérielles modélisées, le logiciel doit utiliser une API (API pour Application Programming Interface) spécifique, que ce soit au niveau du Système d'Exploitation [GCKR12] ou niveau du HAL (HAL pour Hardware Abstraction Layer) [GHP09].

Généralement la partie fonctionnelle des systèmes de simulation natifs ne dépend pas de l'ISAs de processeurs cibles, ce qui les rend attrayant du point de vue du développement.

D'autre part cela rend le simulation subtilement différent du point de vue du flux de contrôle d'exécution cible, comme les optimisations spécifiques cibles ne peuvent pas être facilement appliquées sur le logiciel natif. En dehors du fait ci-dessus, les techniques de simulation natives introduisent leurs propres problèmes qui les éloignent de l'architecture modélisée. Il s'agit notamment de différences de l'espace d'adressage de la cible par rapport la machine hôte, comme le logiciel natif utilise l'espace d'adressage de la machine hôte pour toutes les références de mémoire. Cela se traduit par des espaces d'adressage contradictoires et qui peuvent se chevaucher entre la machine hôte et les architectures cibles simulées. Certaines interactions entre le matériel et le logiciel deviennent impossibles, comme le DMA (DMA pour Direct Memory Access) vers les régions de mémoire allouées par le logiciel.

Plus la complexité des architectures modélisées augmente (tant au niveau matériel et logiciel), plus les solutions natives sont loin des architectures cibles. Ces situations proviennent généralement lors de la modélisation des machines parallèles en utilisant des techniques natives, c'est-à-dire en utilisant le logiciel compilé nativement pour modéliser l'exécution d'un flux d'instructions parallèles, par exemple, dans le cas des architectures VLIW (VLIW pour Very Long Instruction Word). Ainsi, la modélisation de MPSoC hétérogène contenant des processeurs VLIW ainsi que des GPP est un objectif important. Nous devons nous concentrer sur ces architectures futures à l'aide des solutions natives ainsi que sur des solutions de simulation mixte. La Figure 1.2 illustre ces tendances dans les architectures mpsoc hétérogènes du futur.

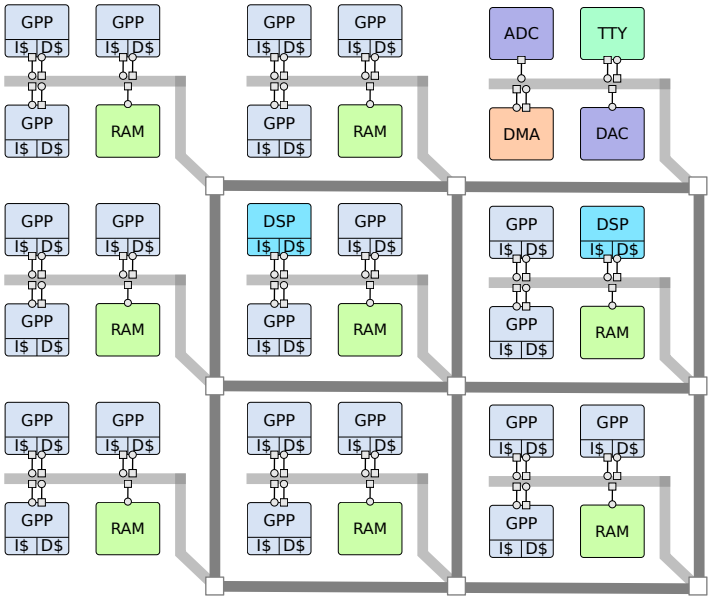


Figure 1.2: MPSoC de future avec plusieurs processeurs GPP et DSP

1.2 Problématique

Simulation native de logiciel est un concept bien connu, allant des techniques primitives basées sur l'exécution directe des algorithmes sur l'hôte pour la vérification fonctionnelle à des implémentations plus avancées reposant sur l'HAL et les notions de liaison dynamique.

Dans sa formulation la plus générale, la simulation du logiciel natifs vise l'exécution

directe du code logiciel sur la machine hôte à l'aide d'une "enveloppe" pour se connecter à un environnement de simulation événementielle. Les propositions initiales suggèrent d'encapsuler le code de l'application dans les modules TLM en utilisant des threads matériels, comme si elles étaient implémentées comme des composants matériels [GCDM92, GYNJ01, BYJ02, WSH08, CHB09a]. Un sous-ensemble des tâches logicielles est encapsulé dans les modules matériel, c'est-à-dire des éléments de traitement de la plate-forme, comme représenté sur la Figure 1.3. Le noyau de simulation matériel ordonance à la fois des fils d'exécution matériels et logiciels, apportant concurrence implicite et non intentionnelle à des tâches logicielles. Effectivement, cela ne permet pas de rendre compte du comportement du système d'exploitation.

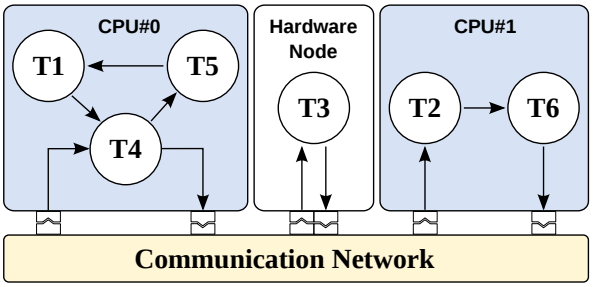
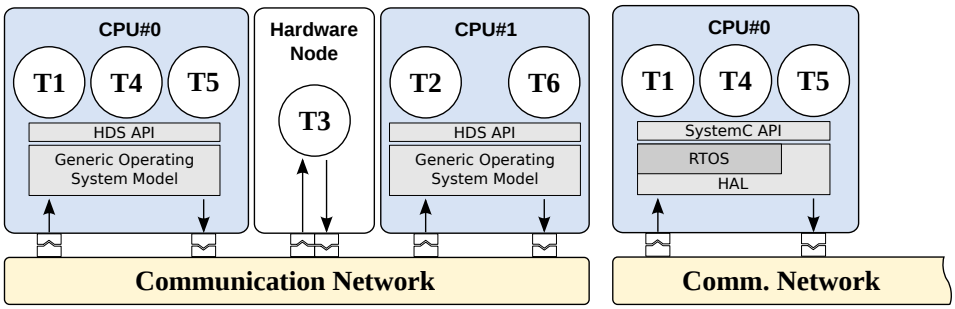


Figure 1.3: Encapsulation du logiciel pour la simulation native

Ces solutions sont simples mais présentent deux inconvénients graves. Premièrement, le code simulé donne une forme très limité de parallélisme, c'est-à-dire des co-routines. Deuxièmement, puisque le logiciel s'exécute dans un module matériel, toutes les allocations de données par le logiciel sont alloués dans l'espace d'adressage du processus de simulation au lieu de la mémoire de la plate-forme cible simulée. Il n'existe aucun moyen pour un composant de la plate-forme d'accéder à un buffer alloué par le logiciel parce que le buffer n'est pas visible d'elle alors qu'il doit l'être, les adresses de la plate-forme de simulation n'ont aucun lien avec les adresses du processus de simulation. En outre, l'exécution du logiciel est limité dans le contexte matériel, tels que la façon dont il accède aux ressources de la plate-forme sous-jacente à l'aide des interfaces de la plateforme par exemple en utilisant les ports dans les modèles TLM (TLM pour Transaction Level Modeling). Ces approches ne sont clairement pas aptes à supporter le code pré-existant.

En raison de la complexité du logiciel embarqué et des exigences de dépendance d'exécution, l'intégration de modèles abstraits du système d'exploitation directement dans les environnements de simulation natifs a été proposé. L'objectif est de fournir une implémentation légère d'un système d'exploitation en utilisant les primitives basées sur des événements de l'environnement de simulation, ainsi chaque tâche logicielle devient un module matériel. En utilisant cette approche, le RTOS (RTOS pour Real-Time Operating System) modélisé repose sur l'ordonnanceur du simulateur de matériel au lieu de l'ordonnanceur que le RTOS aurait utilisé, même si certaines solutions suggèrent de modifier le noyau de simulation de matériel à cet effet. Ces travaux rendent compte des algorithmes d'ordonnancement de RTOS à l'intérieur des de threads matériels simulés pour exécuter des tâches d'application, avec diffèrents contraintes et des priorités. Un ensemble d'APIs HDS (HDS pour Hardware Dependent Software) y compris la création des tâches, IPC (IPC pour Inter-Process Communication) et services de sémaphores sont mises en oeuvre pour s'adapter aux exigences de tâches d'application, comme montre le Figure 1.4.

Malheureusement, ces modèles de système d'exploitation ne sont pas suffisamment détaillés car tous les appels de la bibliothèque C (entre autres), y compris les fonctions de gestion de la mémoire, sont hors du contrôle du modèle de système d'exploitation. Cette limitation rend le développement des pilotes des périphériques impossible en utilisants ces modèles. De même, ces modèles nécessitent la réécriture du logiciel d'application, ils empêchent effectivement l'utilisation du code "legacy".



(a) Avec un modèle abstrait du système d'exploitation (b) Avec un RTOS réel

Figure 1.4: Encapsulation du logiciel avec un système d'exploitation abstrait ou réal

Pour améliorer le réalisme de l'exécution du logiciel, certains travaux ont proposé d'utiliser un RTOS réel au lieu d'un modèle de systèmes d'exploitation abstrait, comme montré dans la Figure 1.4(b). La pile logicielle est toujours encapsulée dans les modèles de matériel, mais le système d'exploitation est le même que celui de la plate-forme réelle. Celui donne certains avantages, par exemple les fils logiciels sont ordonnancés par le RTOS réel, fournissant une modélisation réaliste des propriétés des logiciels temps-réel. De même, ces systèmes fournissent des moyens faciles pour l'exploration de l'architecture comme ils prennent en charge différents niveaux d'abstraction et permettent la modification des types de composants du matériel au logiciel et vice versa. Enfin, ces modèles permettent l'exécution et la validation d'une certaine quantité de logiciel en utilisant la simulation native.

1.2.1 Interfaces Matériel Logiciel

L'abstraction est définie comme un processus de simplification où seules les détails essentiels d'une entité complexe sont maintenus, pour un objectif spécifique. La notion d'abstraction peut être appliquée au logiciel, ainsi que des composants de matériel d'un système MPSoC donné. Une interface matérielle / logicielle sert comme une machine virtuelle, où elle exécute des commandes, par exemple, des instructions, des fonctions, etc, depuis le logiciel et interagit correctement avec la plate-forme matérielle. Des nombreuses interfaces matérielles / logicielles sont possibles dans des environnements de simulation MPSoC [BYJ04, SGP08, GGP08]. Nous nous focalisons sur le niveau TLM pour les composants matériels et comparons diffèrents niveaux d'abstraction du logiciel dans les plates-formes TA (TA pour Transaction Accurate) ainsi que leurs interfaces c'est-à-dire HDS ❶, HAL ❷ et ISA ❸, comme presenté dans la Figure 1.5.

Les couches du logiciel bénéficent du fait que la plate-forme matérielle peut fournir une véritable API pour interagir avec le monde du logiciel. Cette API définit le niveau de l'interface et est généralement mis en oeuvre à l'intérieur des modèles de matériel en utilisant un langage de programmation de logiciels. Cette interface d'API rend les couches logicielles

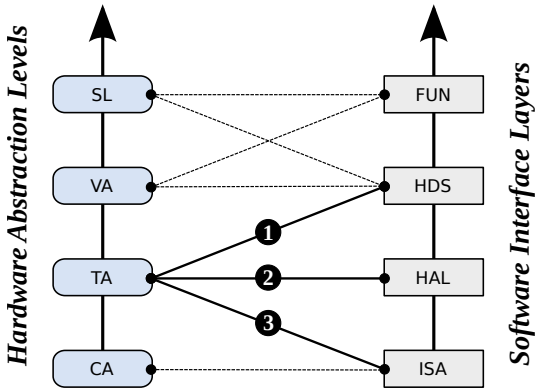
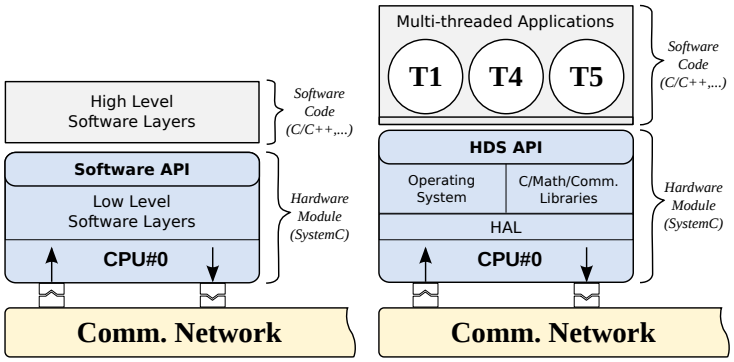
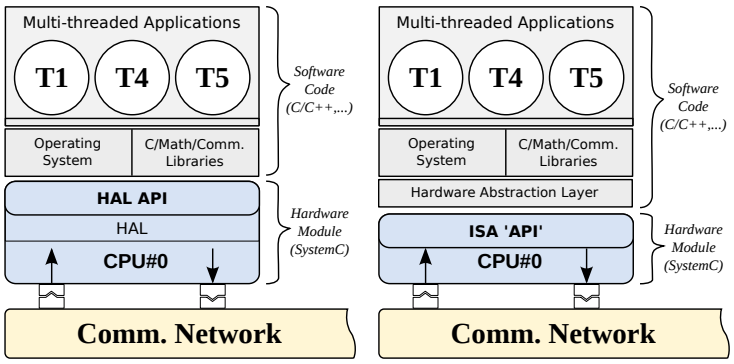


Figure 1.5: Niveaux d'abstraction matérielle et couches d'interface du logiciel



(a) Principe d'exécution en couche (b) Exécution sur la couche d'HDS



(c) Exécution sur la couche d'HAL (d) Exécution sur la couche d'ISA

Figure 1.6: L'exécution en couches du logiciel à différents niveaux de l'interface

supérieures complètement indépendants des modèles matériels inférieurs, car ils peuvent être compilés et exécutés sur l'interface fournie. La Figure 1.6(a) montre le principe de ces modèles d'exécution.

Les modèles de logiciels à couches les plus fréquemment utilisés [BBY+05, TRKA07, PJ07, CSC+09] implémentent l'API HDS pour des applications logicielles de haut niveau. Il s'agit d'une solution difficile car, pour construire des applications réalistes en utilisant cette interface de haut niveau, les modèles matériels doivent mettre en oeuvre de nombreuses APIs.

La quantité de logiciels qui pourraient être validée en utilisant ces interfaces est très limitée, ce qui reduit leur utilisation en pratique. Aucune de ces approches ne cible la création dynamique des tâches ou de migration des threads entre les processeurs, comme cela a lieu sur les plateformes SMP (SMP pour Symmetric Multi-Processor). La Figure 1.6(b) montre l'organisation des plateformes de simulation à la base d'interfaces HDS.

Quelques approches [YJ03, YBB+03, BYJ04, GGP08] reposent sur la définition d'une couche HAL fine qui doit être utilisé pour tous les accès liés au matériel. La couche HAL est mise en oeuvre dans une enveloppe qui comprend un thread matériel pour chaque processeur, appelé l'EU (EU pour Execution Unit). La pile logicielle entière au-dessus du HAL peut être compilée de manière indépendante, y compris le système d'exploitation et les bibliothèques standard et exécuté nativement. Chaque appel de fonction HAL est réalisé dans le contexte d'un EU en supposant que tous EUs appartenant de l'enveloppe partagent le code du système d'exploitation et des structures de données. Comme les fonctions de commutation de contexte appartienent à la couche d'HAL, la migration des threads de logiciel comme dans le SMP est pris en charge dans de telles plates-formes. La Figure 1.6(c) montre la structure des modèles basés sur l'interface d'HAL.

Les plates-formes de simulation natives qui sont basées sur ce qu'on appelle l'API ISA (ou ISA Hôte), comme montre la Figure 1.6(d), n'ont pas été discutées dans la littérature. Les plates-formes TA les plus communes qui assurent un niveau d'exécution à l'ISA du logiciel, cible l'usage des ISSes. La déficience principale de ces modèles ressort de l'utilisation soit des ISSes [LP02, HJ08] ou des solutions à base de DBT [GFP09] (DBT pour Dynamic Binary Translation), qui interprètent / traduisent les instructions au moment de l'exécution et entraînent des simulations plus lentes. L'ensemble des techniques de simulation natives discuté jusqu'ici souffre du problème des espaces d'adressages où le logiciel natif s'exécute dans l'espace d'adressage d'hôte et la plate-forme matérielle simule l'espace d'adressage de la cible. Ces différences rendent difficile la modelisation de certaines interactions entre le matériel et le logiciel, comme expliqué dans la section suivante.

1.2.2 Les Espaces d'Adressage Cible et Hôte

Les plates-formes de simulation natives font face à deux types de dépendances, résultant principalement de la compilation native du logiciel. Les différences dans ISA des processeurs hôte et cible, ainsi que les détails spécifiques des modules matériel doivent être abordées au départ. Ces différences sont résolues en utilisant explicitement l'API HAL pour toutes les interactions entre les composants logiciels et matériels, ce qui donne une pile logicielle indépendante du matériel, à l'exception de la mise en oeuvre de HAL. La deuxième source de dépendance se manifeste par les représentations de la mémoire dans les composants matériels et logiciels. Pour être précis, deux espaces d'adressage différents et parfois contradictoires doivent être considérés, l'espace d'adressage cible (Ⓣ) et hôte (Ⓗ), comme montré dans la Figure 1.7.

Ⓣ La plate-forme matérielle, compilé sur la machine hôte, simule les adresses cibles qui sont connues au moment de la compilation, c'est-à-dire le mappage des adresses dans les composants matériels qui ont été définis à l'avance par le concepteur du système et les décodeurs d'adresse de la plateforme utilisent ces plages d'adresses pour la communication entre les modèles de matériel.

Ⓗ La pile logicielle est compilé pour la machine hôte et toutes les références à la mémoire

sont inconnues au moment de la compilation. Les adresses du logiciel sont connues à l'exécution lorsque la bibliothèque représentant le logiciel est effectivement chargé dans la mémoire de l'hôte. En général c'est le processus SystemC qui charge la bibliothèque et fournit le contexte d'exécution.

Le problème des espaces d'adressage cible et l'hôte n'apparaît pas dans les plates-formes de simulation basées sur l'ISS, où à la fois des composants matériels et logiciels voient le même espace d'adressage, c'est-à-dire l'espace d'adressage cible. Ces différences dans les espaces d'adressage ne permettent pas de modéliser certaines interactions entre les composants matériels et logiciels.

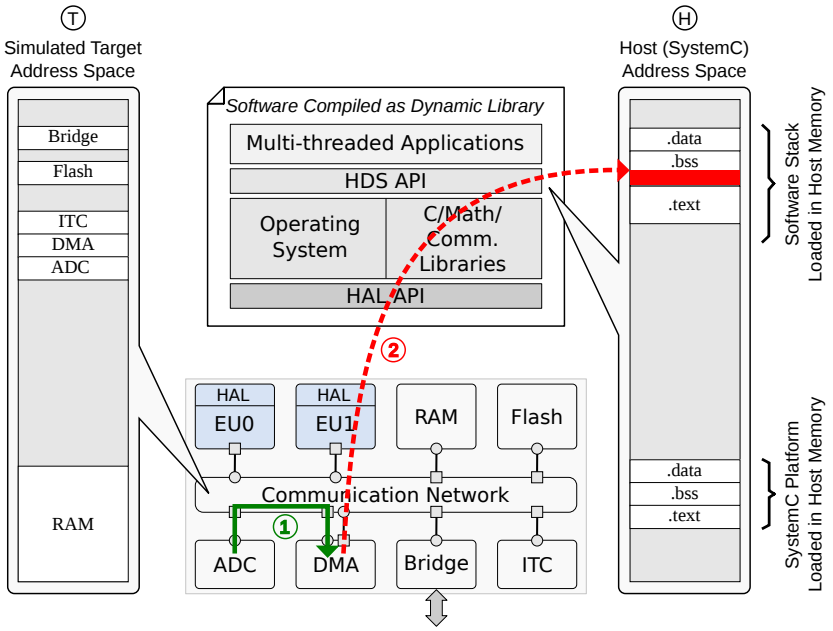


Figure 1.7: Représentations de la mémoire cible et hôte (Adapté de [Ger09])

Deux principales classes de solutions ont été proposées pour résoudre ce problème, le remappage des adresses et l'unification des espaces d'adressages.

1.2.2.1 Les Techniques de Remappage des Adresses

Des techniques simples de remappage effectuent la conversion d'adresses entre l'espace cible et l'espace d'adressage du processus de simulation par l'utilisation de primitives spécifiques pour les entrées / sorties. Cela ne résout pas le problème des accès externes vers des buffers alloués nativement. Les stratégies de remappage plus complexes reposent sur le fait qu'une exception du système d'exploitation de l'hôte sera soulevée lors d'un accès à une mauvais adresse virtuelle.

Le principe est de marquer comme non valides (invalid) toutes les pages mémoires visibles par les composants de la plateforme, en utilisant le système d'exploitation hôte. Tout les accès à ces pages déclenchent une exception que le processus de simulation peut intercepter et gérer. Il peut y avoir des problèmes de performance, si de nombreuses exceptions sont levées, et les aspects techniques de manipulation des chevauchements entre les deux espaces de la mémoire reste un problème. Par exemple, les adresses mémoire au delà de 3 Go

(0xC0000000) dans l'architecture x86 sont réservées pour le code du noyau Linux. La Figure 1.8 met en évidence le principe général des techniques de remappage des adresses.

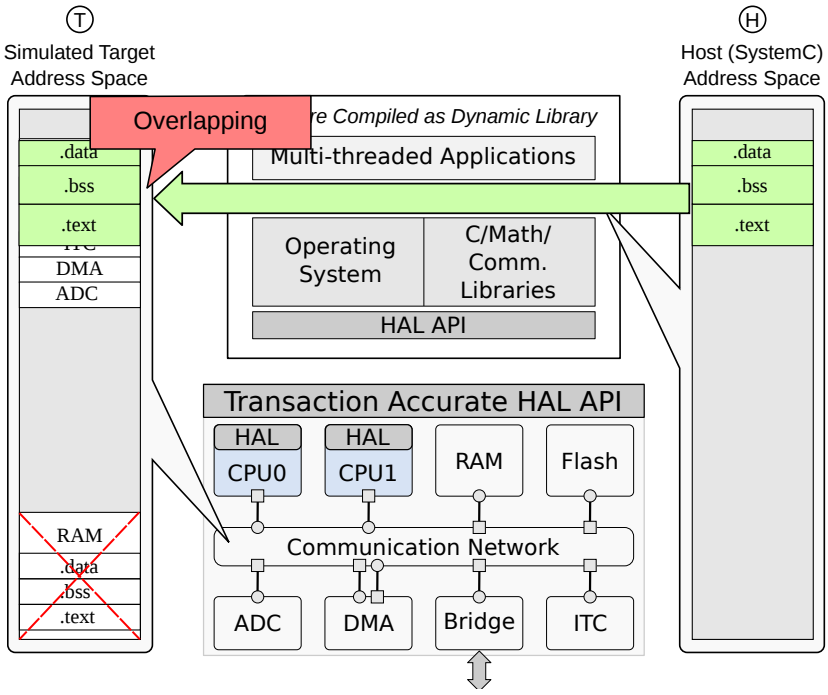


Figure 1.8: Modification de la mémoire cible pour remmapage des adresses

1.2.2.2 Représentation de la Mémoire Uniforme et Edition de Liens Dynamiques

L'unification repose sur l'utilisation d'un mappage de mémoire unique pour le logiciel, et les composants matériels de l'environnement de simulation natifs. Le mappage du processus de simulation est sélectionné à cet effet comme il est également utilisé par la pile logicielle de simulation. L'unification nécessite de modifier la plate-forme matérielle de sorte que chaque composant exporte un ensemble de symboles obligatoires qui doivent être résolus au moment de la liaison pour effectuer un remappage à faible coût. Ces plates-formes requièrent également des modifications du système d'exploitation, afin qu'il accède au matériel uniquement par le biais des appels de fonctions HAL et n'utilise jamais des adresses codées en dur. La Figure 1.9 représente le principe de ces techniques.

Les inconvénients comprennent la modification des modèles de simulation pour créer l'espace de mémoire unifié, l'ajout d'une étape de liaison spécifique visible à l'utilisateur, et le port du système d'exploitation sur la couche HAL natif. Afin de simuler la bibliothèque C complète, en particulier les fonctions de gestion de la mémoire, un espace d'adressage bien séparés doit être fourni au système d'exploitation s'exécutant de façon native.

1.3 Virtualisation Assistée par le Matériel

La virtualisation des ressources informatiques physiques est un concept bien connu [Gol74, Cre81] et fournit des moyens pour le partage de ces ressources pour améliorer l'utilisation du système. Il est similaire à l'abstraction, mais les détails des ressources sous-jacentes ne sont

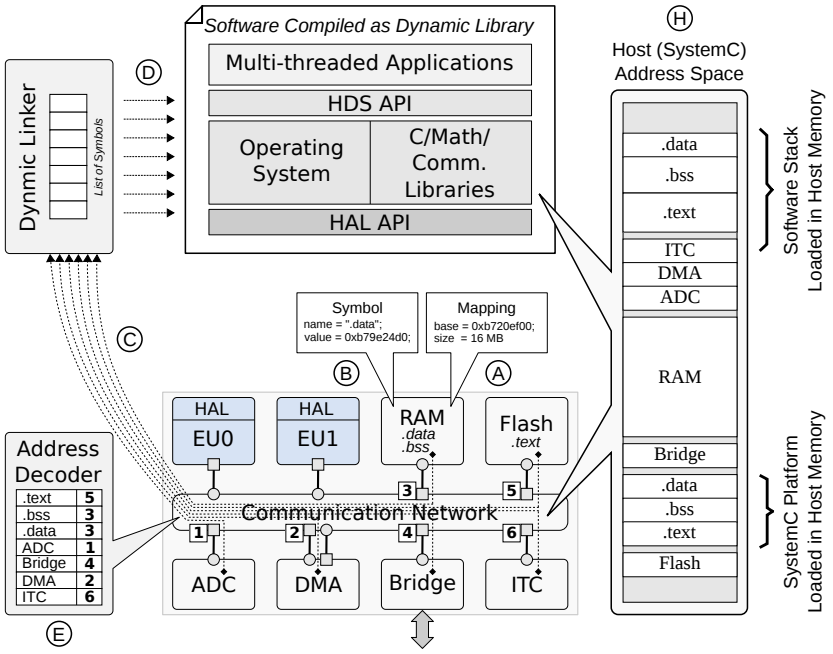


Figure 1.9: Représentation uniforme de la mémoire dans l'espace d'adressage de machine hôte

pas nécessairement cachés pour le logiciel, car il traite de la création de structures logiques qui fonctionnent comme la machine physique réelle.

Dans un système non virtualisé, un seul système d'exploitation contrôle des ressources matérielles alors que dans un environnement virtualisé une nouvelle couche logicielle est introduite, connu sous le nom VMM (VMM pour Virtual Machine Monitor) ou hyperviseur, qui contrôle et arbitre les accès aux ressources de la plateforme. Cela permet l'utilisation de plusieurs systèmes d'exploitation sur une seule machine matérielle, communément connus en tant qu'invités de la VMM. La VMM présente un ensemble d'interfaces virtuelle qui constituent une machine virtuelle à chaque système d'exploitation invité, en leur faisant croire qu'ils ont un contrôle total sur la machine "physique". La terme hôte est couramment utilisé pour désigner le contexte d'exécution de VMM ou le système d'exploitation hôte.

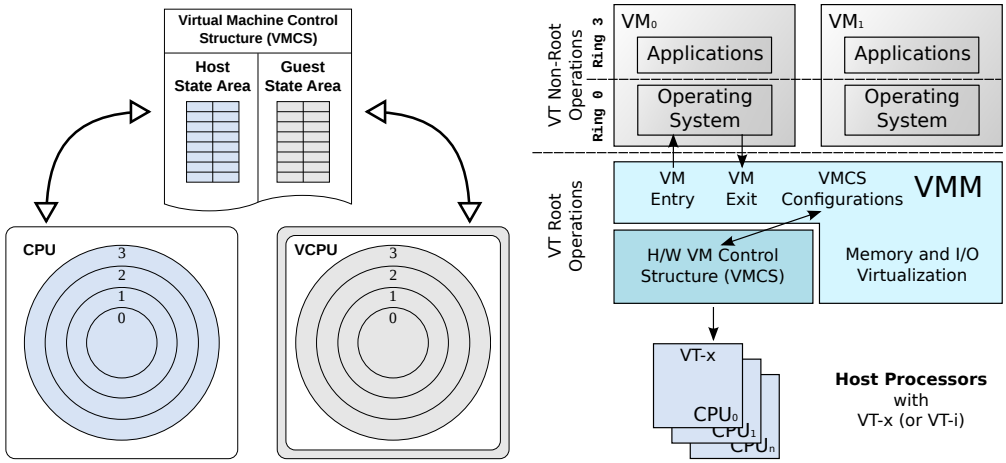
Cette section présente la contribution principale de cet thèse et presente l'utilisation du HAV dans le cadre d'environnements de simulation natifs événementiels pour la modélisation des architectures MPSoC. La technologie d'HAV fournit du matériel dédié à la virtualization et prend en charge les fonctionnalités clés suivantes:

- **Nouveau mode de fonctionnement invité**: Ce mode offre un nouveau contexte d'exécution de la machine hôte dans lequel l'espace d'adressage peut être entièrement adapté. Le logiciel invité s'exécute en mode de fonctionnement non-root alors que le logiciel hôte et VMM s'exécutent en mode de fonctionnement root. La Figure 1.10(a) montre les modes de fonctionnement invité et hôte.

- **Commutation de contexte basée sur le matériel**: Une prise en charge matérielle a été introduite pour la commutation atomique entre les modes hôte et invité et vice-versa, de manière complète et efficace. Le matériel commute les registres de contrôle, les registres de segments et le pointeur d'instruction de telle sorte que la commutation des espaces d'adressage et de transfert de contrôle sont effectué de façon atomique. La

Figure 1.10(b) montre les transitions de VM Entry et VM Exit entre les modes hôte et invités.

- **Rapport de changement de mode invité à hôte**: Chaque fois que le logiciel invité quitte le mode invité, il rend compte du motif de la sortie à la VMM, qui utilise cette information pour prendre une action correspondant.



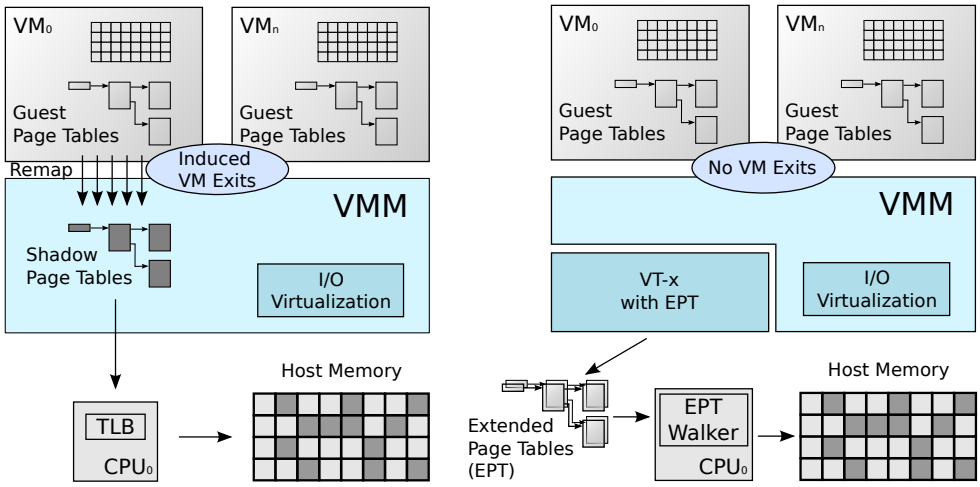
(a) La mode hôte *vs*. invité dans HAV (b) VMCS, operations VT root et non-root

Figure 1.10: Modes hôte et invité dans HAV (Les processeurs Intel)

Chaque transition VMX entre l'hôte et l'invité peut commuter des espaces d'adressage. Cette fonction permet au logiciel invité d'utiliser son espace d'adressage complet, qui peut être différent de celui de l'hôte et de la VMM. L'espace d'adressage invité fait partie de l'espace d'adressage utilisateur sur la machine hôte, qui définit des traductions des adresses physiques de l'invité vers les adresses virtuelles de l'hôte. La VMM fournit les traductions d'adresses suivantes:

- La traduction de GPA (GPA pour Guest Physical Address) vers HPA (HPA pour Host Physical Address) quand la pagination est désactivée dans mode invité et le logiciel utilise des adresses physiques.
- Les traductions de GVA (GVA pour Guest Virtual Address) à GPA et à HPA quand la pagination est activée en mode invité.
- Les traductions NGVA (NGVA pour Nested Guest Virtual Address) à NGPA (NGPA pour Nested Guest Physical Address) à GPA et à HPA quand l'invité lance son propre invité, et la pagination est activée à la fois dans l'invité et l'invité emboîté.

Les traductions précitées sont fournies en utilisant soit les SPT (SPT pour Shadow Page Tables) ou en exploitant le support matériel de pagination à deux dimensions appelé EPT (EPT pour Extended Page Tables) ou RVI (RVI pour Rapid Virtualization Indexing) par Intel et AMD, respectivement. En essence, la VMM est responsable de l'exposition d'une MMU (MMU pour Memory Management Unit) hôte au logiciel invité, lors de la traduction des



(a) Using Shadow Page Tables (SPT) / Utiliser Shadow Page Tableaux

(b) Using Extended Page Tables / Utilisant Extended Page Tables

Figure 1.11: Support pour la virtualisation de la mémoire dans HAV

adresses virtuelles ou physiques invitées en adresses physiques de l'hôte, en utilisant l'une de ces technologies.

La virtualisation de la mémoire en utilisant SPT est fourni par la VMM basé sur HAV par d'interception des toutes les opérations de pagination du logiciel invité, y compris les défauts de page, l'invalidations des pages et l'accès aux registres de contrôle émulés des invités (CR0, CR2, CR3 et CR4). Essentiellement, toutes les tentatives du logiciel invité d'accéder au matériel de traduction d'adresse sont trappées par la VMM et émulées.

Lorsque le support de pagination à deux dimensions est activé dans l'HAV, les adresses physiques employées dans le mode de fonctionnement non-root *i.e.* invité, sont converties en parcourant un ensemble de structures de pagination EPT. Ces structures sont mises en oeuvre dans le matériel, et un composant matériel *i.e.* EPT Page Walker traduit les GPAs à HPAs finaux qui sont utilisés pour accéder à la mémoire de l'hôte. Les Figure 1.11(a) et Figure 1.11(b) montrent les principes génériques des pages shadow et des tables de pages étendues.

1.4 Simulation Native à l'Aide de la Virtualisation Assistée par le Matériel

Nous résoudrons le problème des espaces d'adressage cible et hôte dans la simulation native en introduisant une couche de traduction d'adresses transparente et basée sur le support matériel fourni par la technologie d'HAV. Cette technique de traduction est différente des ISSes basés sur DBT (DBT pour Dynamic Binary Translation), qui invoquent des fonctions de traduction d'adresses à base de logiciel pour chaque accès mémoire, dégradant ainsi la performance de simulation. Cet section donne un aperçu de la façon dont on résout le problème des espaces d'adressage en intégrant une VMM dans un environnement de simulation basée sur les événements.

Concrètement, nous intégrons l'open source KVM basé sur Linux dans l'environnement

SystemC. Notre contribution n'est pas de créer une nouvelle machine virtuelle fondamentale basé sur HAV, mais d'utiliser un existant et de l'intégrer dans un environnement de simulation SoC événementiels pour résoudre le problème des espaces d'adressage décrit dans la section Section 1.2.2.

Les NPU (NPU pour Native Processing Units) sont basées sur le concept de module SystemC et constituent la base de notre framework de simulation. Chaque NPU modélise un processeur natif et fournit l'interface entre les composants SystemC matériels et KVM. Pour l'interfaçage avec KVM, il utilise la bibliothèque de l'espace utilisateur KVM, qui exporte les fonctions clés pour la manipulation de la machine virtuelle. Le module de noyau KVM expose l'ISA du processeur hôte à la pile logicielle, c'est-à-dire l'ISA x86 dans notre cas. La pile logiciel, y compris le système d'exploitation embarqué, la couche d'HAL et l'application est compilée au format binaire hôte et exécutée dans une machine virtuelle.

Chaque processeur natif comprend un fil SystemC pour modéliser le processeur et les composants pour interagir avec la bibliothèque de l'espace utilisateur KVM, tels que les fonctions de rappel et de fonctionnalités pour exploiter l'interface de la bibliothèque de KVM. Les plate-formes de simulation natives demandent certains services à KVM, tels que la création d'une nouvelle machine virtuelle comprenant un ou plusieurs VCPU (VCPU pour Virtual CPU), l'initialisation de l'espace mémoire de l'invité et le lancement d'exécution du logiciel. Chacune de ces demandes est envoyée au module KVM en utilisant l'interface ioctl fournies par le noyau Linux. Chaque ioctl retourne une valeur pour indiquer le succès ou l'échec du service demandé. La Figure 1.12 fournit une vue de haut niveau des unités de traitement natifs et leur relation à KVM et la Figure 1.13 donne le flux global d'exécution de la plate-forme de simulation natifs proposée.

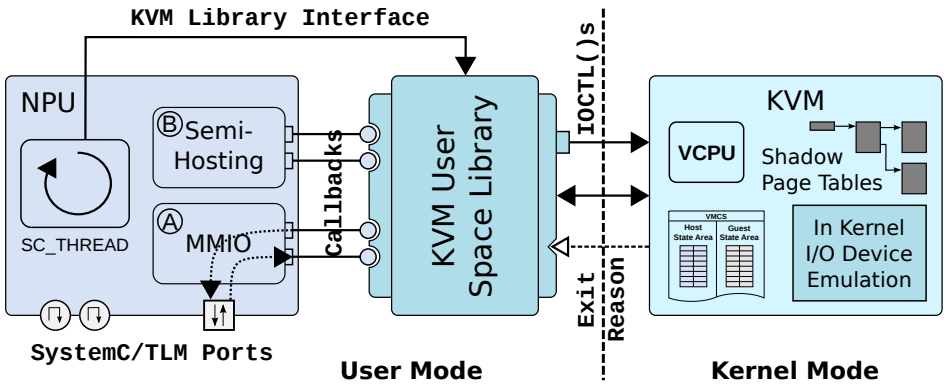


Figure 1.12: Les unités de traitement natifs (NPU) et leur interfaces avec KVM en utilisant la bibliothèque utilisateur

Les plates-formes de simulation natives font face à deux types de dépendances, résultant principalement de la compilation native du logiciel. Les différences dans l'ISA des processeurs hôtes et cible, ainsi que les détails spécifiques de modules matériel doivent être abordés au départ. Ces différences sont résolues en utilisant explicitement l'API HAL pour toutes les interactions entre les composants matériels et le logiciel. Il en résulte une pile logicielle indépendante du matériel, à l'exception de la mise en oeuvre de la couche HAL. En conséquence, notre approche est similaire à la paravirtualisation. La couche d'HAL fournit une interface qui est spécifique au système d'exploitation s'exécutant sur le dessus de celui-ci ainsi que met en oeuvre l'interface HAL en fonction de la plate-forme sous-jacente. Nous avons mis en place notre approche basée sur HAV pour le DNA-OS [GP09]. La deuxième

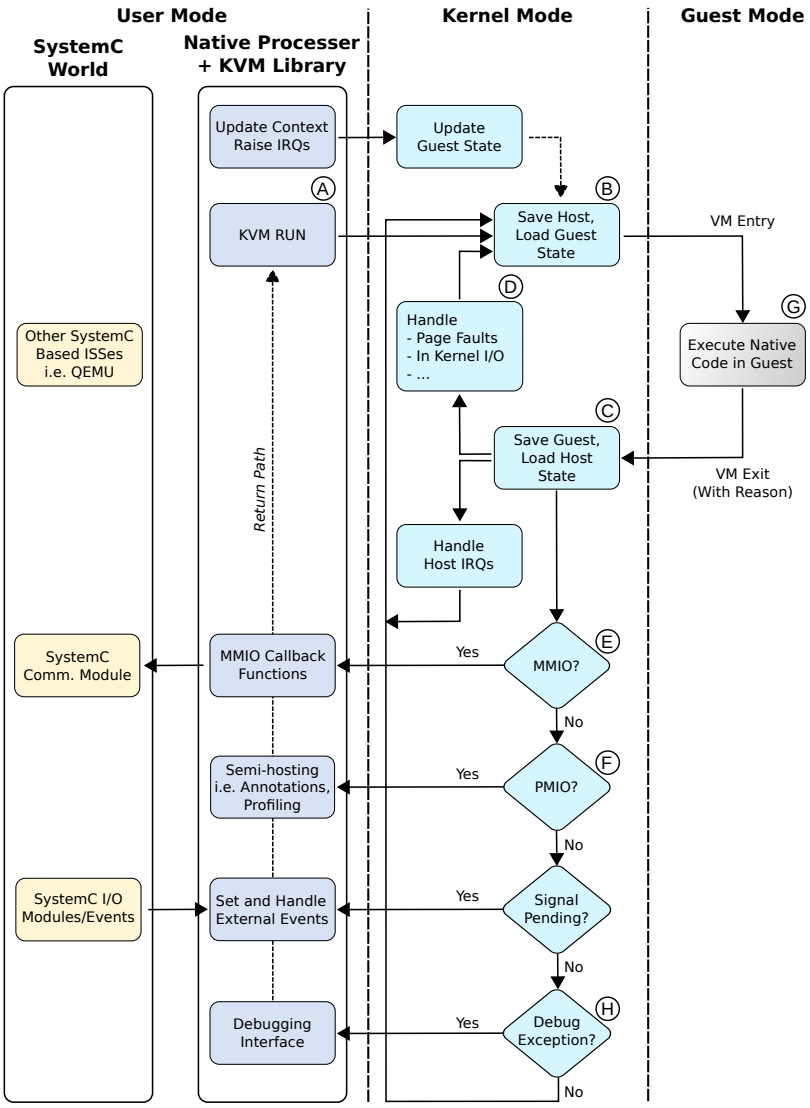


Figure 1.13: Le flux d'exécution dans la simulation native basée sur la virtualisation à l'assistance matérielle

source de dépendances se manifeste par les représentations de la mémoire dans le matériel et les composants logiciels, comme déjà discuté dans la Section 1.2.2. Comme nous basons notre solution sur la technologie d'HAV, qui exporte l'ISA spécifique de l'hôte à ses invités, la couche de d'HAL est implémentée en utilisant l'ISA hôte (x86 dans notre cas).

En utilisant la technologie d'HAV, toutes les interactions entre les composants logiciels et matériels ont lieu au niveau ISA hôte, comme cela a été discuté dans Section 1.2.1 (Figure 1.6(d)). La couche d'HAL fournit les APIs pour gérer les contextes de processus, les primitives de synchronisation, endianness, la mémoire et les accès entrée/sortié et la gestion des interruptions. La Figure 1.14 indique le fait que l'ensemble des couches logicielles supérieures d'HAL sont indépendantes de la plate-forme sous-jacente, et que l'HAL est spécifique à l'architecture de la machine hôte.

Une fois l'exploration initiale de espace de conception terminée pour un SoC donné, la

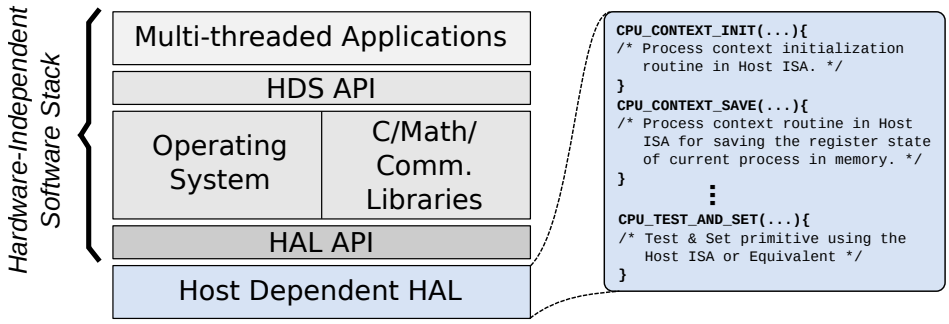


Figure 1.14: Couche d'abstraction matérielle dépendant de la machine hôte

même pile logicielle peut être re-compilée pour l'architecture cible sauf la couche HAL et remplacée par une mise en oeuvre spécifique à la cible. Cela permet de valider une certaine quantité de logiciel pour l'architecture cible, y compris le système d'exploitation et toutes les couches logicielles au-dessus. Comme l'initialisation du contexte et des fonctions de commutation sont inclus dans la couche d'HAL, la modélisation de la création de tâches dynamique et de migration de threads dans les architectures SMP est possible - une condition essentielle dans les systèmes sur puce récents.

La Figure 1.15 est empruntée de [KKL+07] et modifiée pour se concentrer sur la façon dont à la fois la mémoire et entrées/sorties du système cible sont mappées dans l'espace d'adressage de l'utilisateur, ce qui les rend accessibles au modèles de mémoire SystemC. Pour la pile logicielle simulée qui ne connaît que l'espace d'adressage de la cible, les adresses mémoire "physiques" utilisées par le binaire cible sont les adresses virtuelles mappées par le module KVM du noyau Linux à une série de pages réelles de mémoire physique de la plate-forme hôte. Tous les modèles SystemC peuvent accéder à ces pages physiques en utilisant un autre mappage de la MMU qui est également maintenu par le module KVM. L'accès à ces pages est entièrement transparent, c'est-à-dire qu'une application qui s'exécute en mode invité restera en mode invité. Cela conduit à une partage optimale des données bi-directionnelles entre les binaires cible et de l'environnement SystemC.

En outre les accès mémoire, le logiciel embarqué cible doit également accéder aux périphériques d'entrée/sortie. Le MMIO (MMIO pour Memory-Mapped I/O) et le PMIO (PMIO pour Port-Mapped I/O) sont les deux méthodes bien connues qu'un processeur peut utiliser pour effectuer des accès entrées/sorties. Comme l'architecture x86 supporte les deux méthodes, le logiciel cible simulé peut accéder aux périphériques d'entrée/sortie avec ces deux méthodes. Cependant, puisque la plupart des processeurs embarqués ne supportent que MMIO, comme ARM par exemple, nous supposons que les composants matériels sont accessibles par MMIO seulement.

Par opposition à l'accès mémoire, le MMIO ne peut pas être mappé et être directement accessible car le comportement de lecture / écriture d'accès à un registre matériel est normalement différent de celui de la mémoire. Un accès registre du matériel peut déclencher certaines actions du composant matériel cible au lieu de simplement lire / écrire une valeur de données. Comme tous les périphériques d'entrée / sortie d'un MPSoC sont réellement modélisés au sein de SystemC, à la place du vrai matériel, SystemC doit obtenir le contrôle et le transmettre à des composants matériels, chaque fois qu'une instruction de chargement ou stockage est exécuté. Ce processus peut se réaliser naturellement sur une plate-forme de virtualisation basée sur KVM.

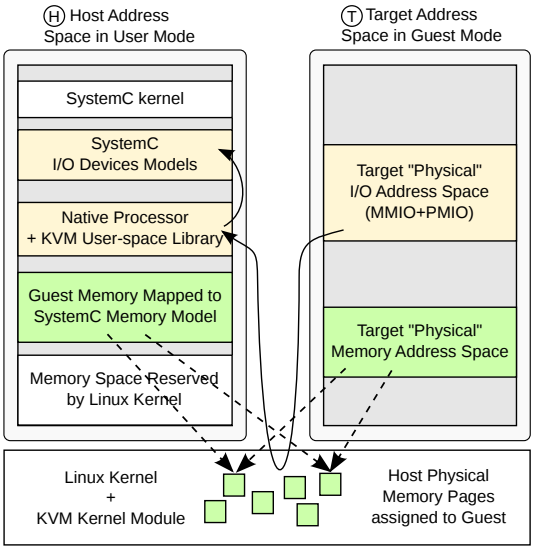


Figure 1.15: Accès mémoire et d'entrées / sorties

Lorsque le logiciel cible simulé effectue l'accès MMIO d'une adresse qui n'appartient pas à l'espace de mémoire, une exception de page est lancée par la MMU du matériel qui impose au processeur de quitter le mode invité et permet au pilote KVM de gèrer l'exception. Le pilote transfère l'accès MMIO à la bibliothèque KVM dans l'espace utilisateur, qui transmet cet accès au processeur natif (NPU), en utilisant les fonctions de rappel installées lors de l'initialisation, comme le montre la Figure 1.13 Ⓔ. A l'aide de l'API KVM, le processeur natif peut obtenir les adresses d'entrée/sortie cible et lancer les opérations d'accès SystemC normales en utilisant les ports TLM et les composants de communication, comme le montre la Figure 1.12 Ⓐ.

Comme le logiciel invité utilise des adresses d'entrée/sortie cibles qui sont exactement les mêmes que celle simulées par la plateforme SystemC, les composants de communication peuvent utiliser des tables de décodage d'adresses attribuées de manière statique. Cet aspect particulier nous donne la liberté d'utiliser des modèles de matériel non modifiés, par opposition à la technique proposée dans [Ger09, GHP09], où les composants matériels doivent être modifiés pour supporter la liaison dynamique, comme montre la Figure 1.9. Grace à l'attribution des adresses statiques dans le décodeur de la plate-forme, la pile logicielle compilé statiquement peut être utilisée, et ne nécessite pas de support de liaison des composants de la plateforme à l'execution.

La solution globale se compose d'une couche HAL dependant de machine hôte, le module KVM de noyau Linux, la bibliothèque KVM en espace utilisateur et les modèles de processeurs natifs basés sur SystemC, comme le montre la Figure 1.16. Deux points importants doivent être revus; tout d'abord, l'espace d'adressage de la mémoire visible par le logiciel invité est le même que celui mappé par le modèle de mémoire SystemC et il est accessible de façon transparente au logiciel invité. En second lieu, les accès d'entrée/sortie initiés par le logiciel embarqué passent par le module KVM du noyau et une bibliothèque dans l'espace utilisateur pour atteindre le modèle de processeur natif, ce qui lance les transactions réelles de lecture/écriture sur le réseau de communication de la plate-forme matérielle.

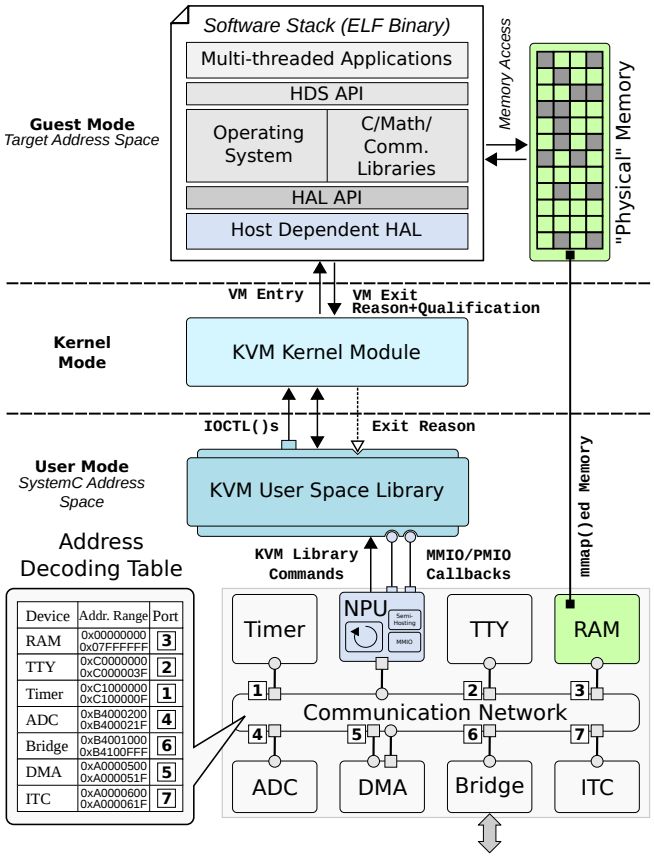


Figure 1.16: Processeur natives, bibliothèque KVM, KVM et la pile logicielle invité

1.5 Conclusion

Ce chapitre a présenté la contribution principale de cette thèse. Nous avons démontré que le problème des espaces d'adressage introduit par les techniques natives peut être résolu en utilisant la technologie HAV. Les éléments clés de la solution proposée sont les NPUs, une couche d'HAL dépendant d'hôte et des accès entrée / sortie à l'aide de MMIO fournie par KVM.

La solution proposée résout efficacement le problème des espaces d'adressage cible et hôte. Elle permet d'utiliser le matériel et des composants logiciels non modifiés. En outre, elle permet de valider une certain quantité de logiciel au dessus de la couche d'HAL et n'impose pas de contraintes de codage du logiciel. La solution proposée est évolutive et peut être utilisé pour la simulation de plusieurs éléments de traitement, et en même temps flexible pour les modèles de plate-formes de simulation y compris de plate-formes hybrides (avec d'autres technologies d'exécution du logiciel) et des modèles de mémoire partagée. La prise en charge des techniques d'instrumentation automatique est également possible dans la technique proposée.

Certaines limitations comprennent l'utilisation de la couche d'HAL et la nécessité pour toutes les interactions de logiciel/matériel de passer à travers de cette couche. Les opérations d'entrée/sortie nécessitent la commutation de mode invité à mode hôte, dégradant ainsi la performance des applications intensives en entrée/sortie. La prise en charge de code auto-modifiant n'est pas disponible dans notre solution, comme la pile logicielle est compilée

statiquement et aucune aide pour la traduction à l'exécution est disponible. La précision de la simulation du point de vue de l'estimation de la performance est très dépendante de l'exactitude de l'annotation; ainsi l'estimation de la performance des architectures complexes est difficile en utilisant des techniques d'annotation traditionnelles. Bien que la solution proposée présente certaines limitations, l'intérêt de cette approche est indiscutable.

Part II

Unabridged English Version

A dream doesn't become reality through magic; it takes sweat, determination and hard work.

Colin Powell

2 Introduction

The current state-of-the-art in *Very-Large-Scale Integration* (**VLSI**) technology allows the fabrication of several billion transistors on a single chip. Designing and integrating hardware with such a huge number of transistors is very difficult. One simple solution for the hardware designers, is to put many relatively simple processors on a single chip rather than designing a complex uniprocessor machine. Moreover, the scalability of mono-processors is constrained by the power dissipation factors, as one cannot simply increase the clock speed to gain performance benefits.

High-Performance Computing field is going towards multi-core or *Chip Multiprocessor* (**CMP**) with dozens of high performance *General Purpose Processors* (**GPPs**). In the integrated systems field, power and yield issues require the use of simpler architectures. These include processors that feature a higher MIPS to Watt ratio, as well as specialized hardware IPs. This trend continues towards more regular and homogeneous structures but still including specific hardware and possibly instruction set extensions in processing elements. These multiprocessors are specific to a design class known as *Multi-Processor System-on-Chip* (**MPSoC**). Many industrial sectors, such as telecommunications, aerospace, automotive, military and consumer electronics are currently using such systems.

2.1 The Hardware/Software Co-Design Challenge

The MPSoC design complexity is increasing due to the number and type of processing elements found in such systems. According to ITRS, as many as 6000 processors are expected on a single SoC by the end of year 2026, as shown in Figure 2.1. Designing and verifying a hardware solution with such high number of processing elements, with even more complex software including hundreds of applications to exploit it, is an enormous challenge.

Similarly, time-to-market is of prime importance in consumer electronics life-cycle, for product manufacturers to remain competitive and be successful in meeting market demands. The ever longer design and fabrication delays of MPSoC systems undermine this goal, as software development depends on hardware availability. Virtual platforms *i.e.* simulation

models provide a feasible alternative, as software development can start early and in parallel with hardware design. Moreover, software teams need simulation platforms, even if the hardware is available, as early prototypes are usually very expensive.

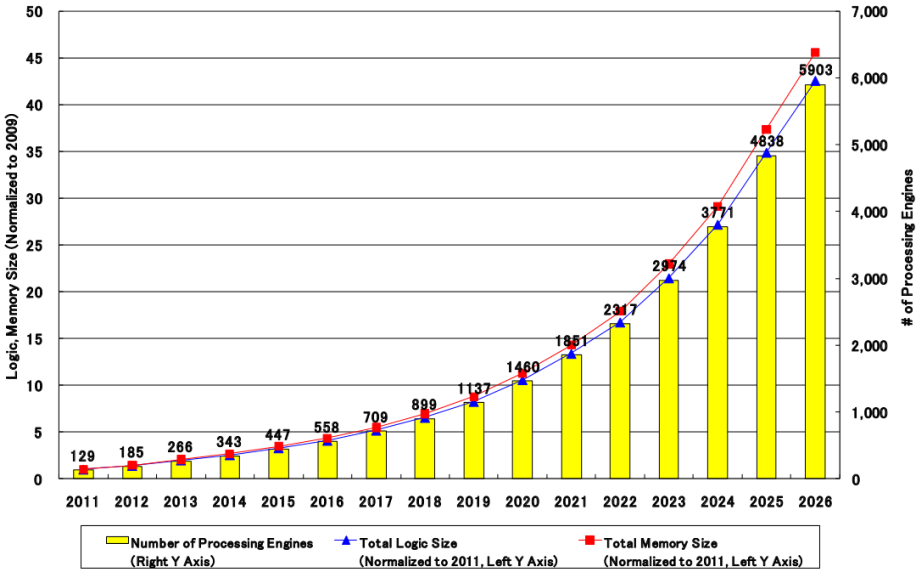


Figure 2.1: SoC Consumer Portable Design Complexity Trends [ITR]

Simulation relies on models and as it can be understood that if the models are close to reality, simulation will be slow, and if they are abstract, the accuracy of results may be disputable. Several models exist for execution of software in simulation: the first one relies on *Instruction Set Simulators* (**ISSes**) that are the most mature and commonly used processor virtualization technology. ISSes execute the cross-compiled target binaries (Operating System + Applications) and mimic the behavior of target processors by using instruction accurate interpretation [Bel73]. Their principle drawback comes from the very slow simulation speed. The second one relies on emulation (mostly) through *Dynamic Binary Translation* (**DBT**) [Bel05] but requires relatively high development cost. The third one uses native simulation *i.e.* code execution without cross-compilation. Native simulation techniques [BBY+05, JBP06] achieve higher simulation speed as they overcome the interpretation/translation cost, by compiling the embedded software to the host binary format and executing it directly on the host processors. In order to access the modeled hardware resources, the software must use a specific *Application Programming Interface* (**API**), either at Operating System level [GCKR12] or *Hardware Abstraction Layer* (**HAL**) level [GHP09]. For a detailed overview and comparison of these software interpretation techniques, kindly refer to [PFG+10].

Usually the functional part of native simulation schemes does not depend on the *Instruction Set Architecture* (**ISA**) of target processors, which makes them attractive from development point of view. On the other hand this makes the simulation subtly different from the target execution control flow perspective, as target specific optimizations cannot be easily applied on the native software. Apart from the above fact, native simulation techniques introduce problems of their own that take these solutions further away from the modeled architecture. These include address space differences from the target, as the native software uses host machine address space for all memory references. This results in

conflicting and possibly overlapping address spaces between the host machine and simulated target architectures. Certain interactions between hardware and software become impossible, such as *Direct Memory Access* (**DMA**) to software allocated memory regions.

As the complexity of modeled architectures increases (both in hardware and software), native solutions lean away from the target architectures and provide far from realistic results. Such situations usually arise when modeling parallel machines using native techniques *i.e.* using natively compiled software to model the execution of a parallel instruction stream, for example, in case of *Very Long Instruction Word* (**VLIW**) architectures. VLIW is an architecture of choice for many DSPs, thanks to the simpler hardware design and sophisticated software support. Examples include the TI C6000 series, Analog Devices SHARC DSPs and TriMedia processors from NXP Semiconductors. Thus, modeling heterogeneous MPSoCs containing VLIW as well as GPPs is a desired goal. We need to focus on such future architectures using native as well as mixed simulation solutions. Figure 2.2 depicts such future trends in heterogeneous MPSoC architectures.

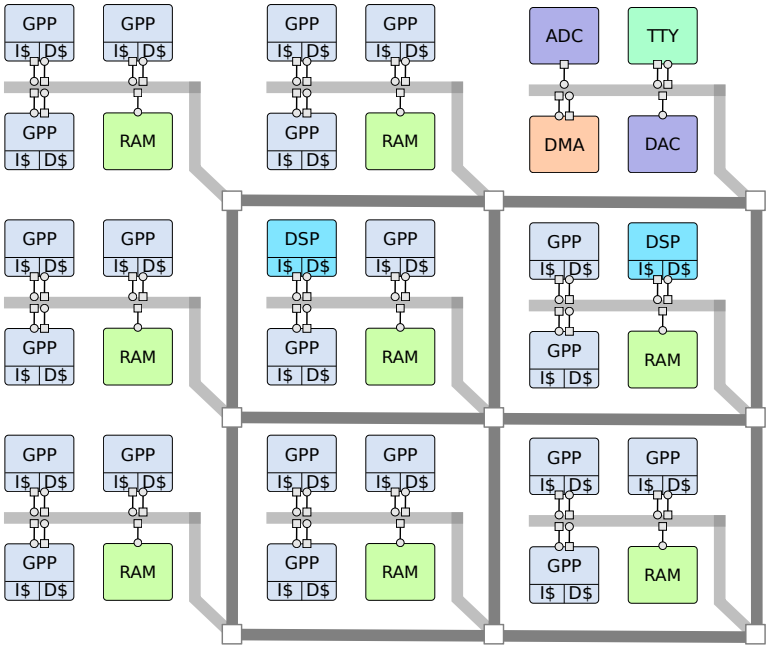


Figure 2.2: Future MPSoCs with Many GPPs along with a few DSPs and Hardware IPs

2.2 Main Contributions and Organization

The primary contribution of this thesis is in native simulation domain and we focus on problems arising from the use of existing native techniques. We propose a novel *Hardware-Assisted Virtualization* (**HAV**) based native simulation solution that removes a decisive limitation of all previously proposed native techniques. We demonstrate the scalability and flexibility of our technique using multiprocessor and hybrid simulation case-studies. For the secondary contribution, we focus our attention on the simulation of VLIW processors using native platforms. Thanks to our HAV based native approach, we investigate and propose a *Static Binary Translation* (**SBT**) flow using a retargetable compiler-based components to achieve fast and accurate simulations.

The rest of this thesis is organized as follows:

- **Chapter 3**: Defines the native simulation technique, its key components and principle problems faced while realizing such a simulation platform. We also describe key issues that must be addressed for simulating a VLIW processor architecture on scalar machines.
- **Chapter 4**: Reviews the state-of-the-art on native simulation techniques as well as binary translation techniques for VLIW processors.
- **Chapter 5**: Explains our main contribution for native simulation of MPSoC using HAV technology. We present the solution to main challenges of native simulation as well as a hybrid simulation platform using different simulation technologies.
- **Chapter 6**: Augments the native simulation technique by a SBT flow for cross-compiled VLIW software binaries. The translation flow is retargetable and generates native code, which is directly executed on top of our HAV-based native platform.
- **Chapter 7**: Manifests a set of experiments and their results to validate the proposed contributions of this thesis.
- **Chapter 8**: Summarizes the key contributions and concludes this work with possible future directions and ideas.

If I had only one hour to save the world, I would spend fifty-five minutes defining the problem, and only five minutes finding the solution.

Albert Einstein

3

Native Simulation of MPSoC: A Retrospective Definition & Problems

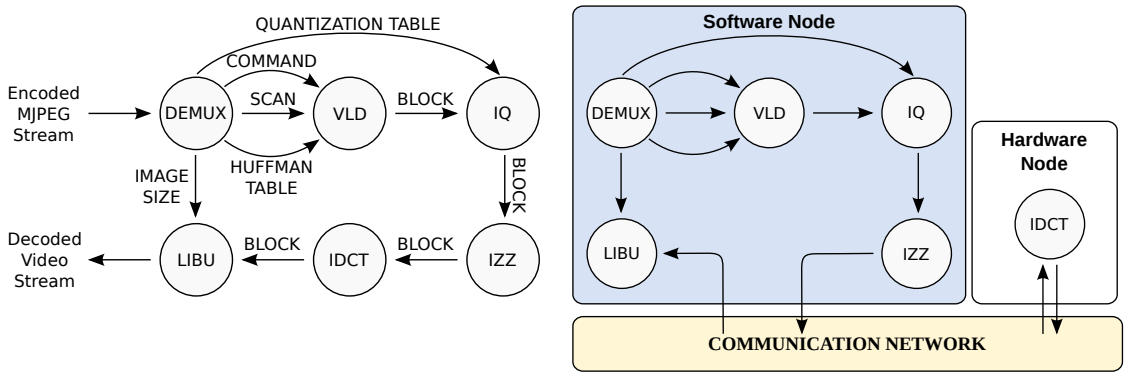
The complexity of current *Multi-Processor System-on-Chip* (**MPSoC**) systems forbid the use of analytical methods for their validation and *Design Space Exploration* (**DSE**). A number of research efforts have established the importance of simulation models in MPSoC contexts. Many of these techniques are based on the principle of native execution, in order to speed-up the software simulation. However, it is difficult to handle all sources of timing dependencies. Nevertheless, these models provide a feasible alternative and are an active area of research.

Simulation models can be developed for different abstraction levels, each answering a different design and precision question. SystemC is an industry standard C++ based library, which is commonly used in modeling and simulation of MPSoC systems and supports multiple abstraction levels. We will elaborate on some of the key features of SystemC and how they relate to our target abstraction level. Another objective of this chapter is to position our work *w.r.t.* these abstraction levels and establish a list of questions that we intend to answer in this thesis. In order to position our work in its context, we start by presenting the architecture of a generic MPSoC system along with some basic concepts necessary for its comprehension.

3.1 Generic Architecture of An MPSoC

A generic MPSoC architecture can be considered as a set of processing nodes that execute the overall tasks of the system. These tasks can be either in software or hardware and task allocation to these nodes can be static or dynamic, depending on the system requirements, such as performance and energy consumption.

A given node is termed as hardware node, if it is non-programmable and is dedicated for a specific purpose such as hardware accelerators. For example a deblocking filter, which improves the visual quality of decoded video frames by smoothing the sharp edges between blocks, could be implemented as a hardware node. Hardware nodes exploit data parallelism and are meant to improve the system performance. Software nodes are programmable and



(a) Specifications of MJPEG Application (Tasks) (b) An Example MPSoC Platform

Figure 3.1: Task Partitioning of the MJPEG Application on an MPSoC Platform

serve as general purpose processing elements in MPSoC platforms. These nodes are actually composed of hardware and software parts, including processor subsystem (CPU, Caches, Memories, Communication Network, *etc.*) and software stack (Applications, *Operating System* (**OS**), Libraries and *Hardware Abstraction Layer* (**HAL**)).

Figure 3.1(a) shows the set of tasks comprising the *Motion-JPEG* (**MJPEG**) application, where an MJPEG stream is decoded using these tasks and decoded video stream is produced, which is subsequently displayed using a Framebuffer component. The *Inverse Discrete Cosine Transform* (**IDCT**) task consumes most of the processing time and is fixed in nature, making it a good candidate for hardware implementation. Figure 3.1(b) shows a possible partitioning of MJPEG application on a generic MPSoC platform where software and hardware tasks execute in parallel and communicate through the on-chip communication network.

3.2 Architecture of Software Nodes

A software node is composed of both hardware and software parts known as Processor Subsystem and Software Stack, respectively, as shown in Figure 3.2. The hardware part includes CPUs, Cache Memories, *Random Access Memory* (**RAM**), Interrupt Controllers, *Direct Memory Access* (**DMA**), Communication Network and other peripherals. Interface components, such as Bridges can be used to communicate with other hardware/software nodes using the global on-chip communication network. Usually the processors in a software node are identical, where they support at-least the same *Instruction Set Architecture* (**ISA**) and execute the same software stack. Such nodes represent *Symmetric Multi-Processor* (**SMP**) architectures. Moreover, these nodes are programmable and provide a great flexibility in MPSoC systems.

The software stack is composed of high level multi-threaded applications, *Operating System* (**OS**), standard libraries (C, Math and Communication *etc.*) and *Hardware Abstraction Layer* (**HAL**). Software applications are usually multi-threaded and concurrently execute multiple tasks (*true* or *false* parallelism, depending on the underlying resources) for providing the overall embedded software functionality. The software stack has two distinct *Application Programming Interface* (**API**) layers *e.g.* the *Hardware Dependent Software* (**HDS**) API [JBP06, DGM09] and HAL API, where the former provides OS-specific functionalities to the applications and later provides processor subsystem specific services to the HDS layer.

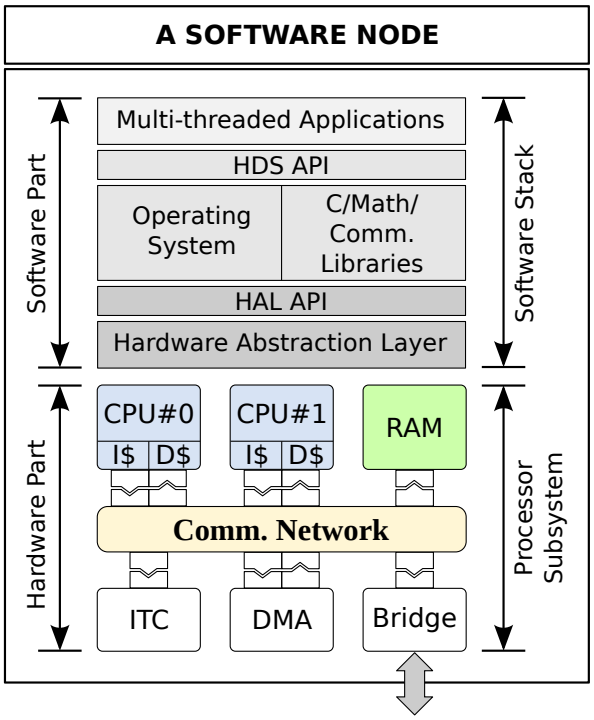


Figure 3.2: Architecture of a Software Node

The HDS provides services for coordinating multiple tasks and shares underlying processor subsystem resources. The HAL layer is responsible for hiding the processor subsystem specific details from the operating system, as to facilitate HDS layer portability across different processor subsystems [YBB+03]. The structure of software stack, as presented here, is the same as in classical computer systems [Tan84]. Such layered software structure could become in-efficient in some specific cases, such as *Digital Signal Processor* (**DSP**) based systems, which usually do not use operating systems.

As discussed above, the processing units in software nodes are identical thus representing SMP systems. *Asymmetric Multi-Processor* (**AMP**) systems can be represented using multiple software nodes where each software node can use different type of processing elements. As an example, we could consider an MPSoC system with two software nodes where the first node contains *General Purpose Processors* (**GPPs**) and the other based on *Very Long Instruction Word* (**VLIW**) DSPs, as shown in Figure 3.3. These software nodes could communicate with each other directly using the global communication network and their own local memories or in-directly via the shared memory.

3.2.1 Key Terms

- **Simulation Platform** : A hardware platform refers to the collection of hardware components including one or more processors and the interconnection network. A platform can be *real* such as a *Field Programmable Gate Array* (**FPGA**) and the final *System-on-Chip* (**SoC**) or *virtual* that is based on a modeling language, allowing us simulate the hardware and execute the software stack on top of it. In this thesis, all of the platforms discussed are virtual and will be subsequently referred as *simulation platforms*.

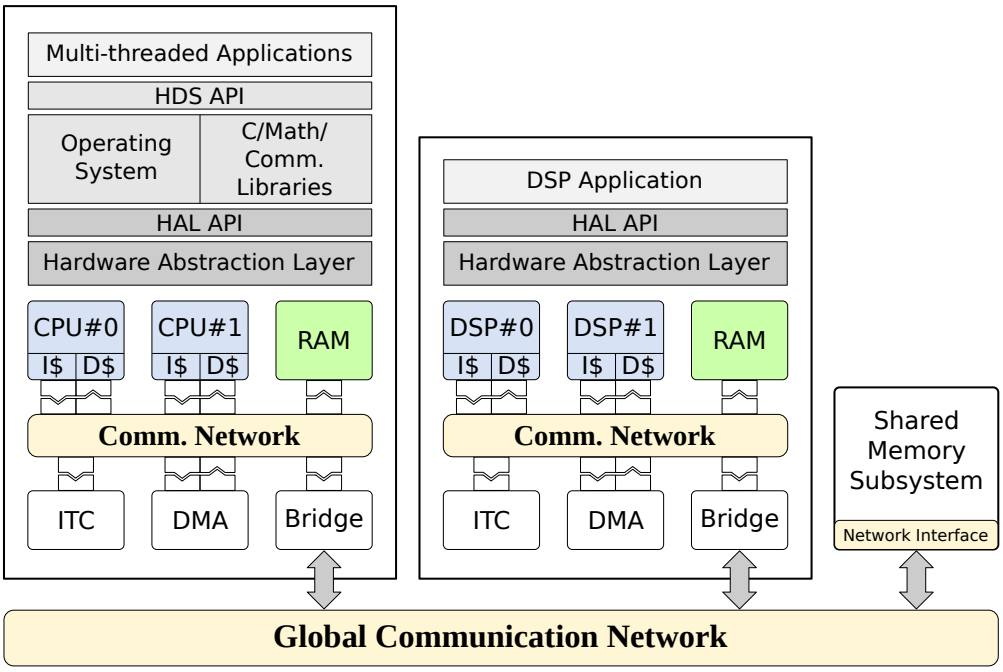


Figure 3.3: Multiple Software Nodes with Different Types of Processors

- **Target *vs.* Host Processor** : The processor that will be actually used in the SoC platform is referred as *target* processor. This includes the most frequently used processors in embedded systems *e.g.* ARM, MIPS, SPARC processors and C6x DSPs. The *host* processors, which execute the simulation platform, are usually different from the target processors and are commonly found in desktop machines *e.g.* x86 series processors from Intel and AMD.

- **Execution *vs.* Native Execution** : The execution of *cross-compiled*[1] software on the target processor will be referred to as *execution*. On the other hand, the execution of software on the host processor, after compilation of source code for the host machine, will be referred to as *native execution*. The execution of software on a target processor, which is simulated by an *Instruction Set Simulator* (ISS) is *not* considered as native execution.

3.3 Description Languages for Simulation Models

Programming languages have been used to describe the hardware component models as to ease the *Design Space Exploration* (DSE) and validation of MPSoC architectures. Principal problems associated with hardware modeling include the parallel semantics of hardware components, their structural information (behavior/interfaces) and modeling of timing behavior in these components, using software languages such as C/C++.

Many research efforts have resulted in new C/C++ based languages, which provide the semantics necessary for hardware validation [KDM90, GL97, VKS00] and even syn-

[1] Cross-compilation refers to the generation of executable code for a platform *other* than the platform on which the compiler is running.

thesis [DM99]. These languages raise the abstraction level of modeled hardware, by ignoring the unnecessary low-level details, enabling the fast architecture exploration of MPSoC systems that would otherwise be *almost* impossible using the lower abstraction levels, such as *Register Transfer Level* (**RTL**) and gate level descriptions.

The last decade has seen the emergence of description languages based on C++, such as SystemC [SYS] and the ones inspired by SystemC itself such as SpecC [GZD+00]. SystemC has become an industry standard and is considered as a principle language for design and validation of *System-on-Chip* (**SoC**). All simulation platforms presented in this thesis are modeled using SystemC.

3.3.1 SystemC: A Modeling Language

The SystemC [SYS] is usually referred as a language just like VHDL and Verilog, but in reality it is a library of C++ classes and macros providing support for the modeling of digital systems. Besides the infrastructure for parallelism, hardware interconnections and time modeling, it also includes an event-driven simulation kernel that allows for the *Discrete Events Simulation* (**DES**) of the modeled hardware. The simulation platform and its components are compiled using a standard C++ compiler and linked to the SystemC library. Thus, a SystemC based model is an executable program and its execution represents the simulation of the modeled system.

SystemC supports the hardware modeling at different abstraction levels, from high-level functional models to the detailed cycle accurate RTL models (*c.f.* Section 3.4). The Accellera Systems Initiative [Acc] merged with OSCI (Open SystemC Initiative) in 2011, is responsible for preparing and disseminating the SystemC specifications and reference implementation.

3.3.1.1 Main Elements

SystemC addresses the modeling of both software and hardware using C++, but our primary interest here is the architecture modeling of hardware components, so we focus on the SystemC features that are specific to this domain. SystemC provides classes for structural and functional modeling of hardware components. We describe the principle elements of SystemC in the following text.

- **Modules and Hierarchy** : A SystemC module is simply a class definition that contains functionality with state, behavior and structure for hierarchical connectivity. SystemC modules are the basic building blocks of SystemC based design hierarchies, which are interconnected using channels. A module contains a variety of elements that make-up its body including constructors, data members, ports, channel instances, modules instances, destructors, processes and helper functions. Only the constructor is required; to be useful a module must contain either a process or other module instance(s).

- **Interfaces** : A SystemC interface is an abstract C++ class that contains no data members and only provides the *pure virtual* declarations of methods referenced by SystemC channels and ports. In other terms, a SystemC interface serves as an API to a set of derived classes, and ensures independence between the definition and implementation of communication mechanisms. This independence enables us to replace a given communication mechanism with another one, provided they implement the same interface without requiring any changes to the rest of system.

- **Channels** : A SystemC channel is a C++ class that implements one or more SystemC interfaces and provides methods for communication between modules.
- **Ports** : A SystemC port is a class templated with and inherits from the SystemC interface(s). Ports allows modules to communicate with other design elements, by forwarding the interface method calls to the corresponding channels.
- **Events and Sensitivity Lists** : SystemC uses processes to model concurrency, and it works in a cooperative multi-tasking fashion where processes are non-preemptive. Each process executes a small chunk of code, voluntarily releases control and lets other processes execute in the same simulated time space. Events are key to such execution models, like in the SystemC simulation kernel. An event is the occurrence of a notification at a single point in time and it does not have a duration or value. A process has to be watching for an event to actually catch it. SystemC allows processes to wait for events using sensitivity lists, which can be static or dynamic. Only two types of actions are possible with events, either a process can wait for it to happen *i.e.* be sensitive to it or a process can notify an event, resulting in the execution of waiting processes.
- **Processes** : The simulation of concurrency in SystemC is the same as in any *Hardware Description Language* (HDL). The processes are the main functional elements in SystemC and have two main types *i.e. methods* and *threads*. As discussed earlier, each process can be sensitive to a set of events. A method could be called many times by the SystemC simulation kernel *i.e.* on every notification of events from its sensitivity list. A method is executed *atomically*, from the simulated time perspective, and it cannot call the SystemC time synchronization `wait()` function.

  Threads on the other hand are called only once by the SystemC kernel, at the start of simulation. Threads can call the SystemC synchronization `wait()` function, and execute *atomically* between two such calls. Once a `wait()` function is invoked, the caller thread is suspended and other process can be executed. A process can notify events, resulting in the execution of *sensitive* methods and resumption of *waiting* threads. A process can also notify timed events where the event will happen in future *w.r.t.* to the simulated time. Once all of the resumed processes have been executed, the SystemC simulation kernel advances the simulation time to the next timed event, and the simulation continues until there are no more processes ready for execution.

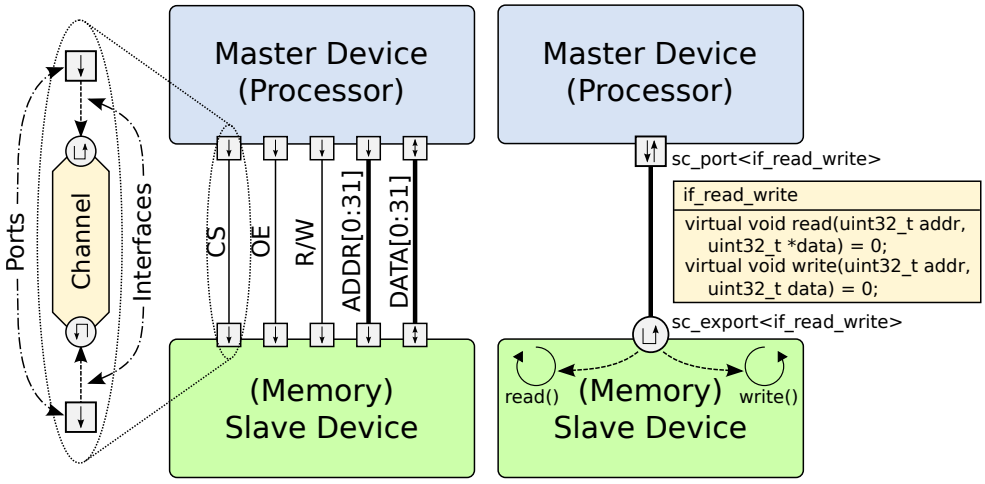
3.3.1.2 Modeling Abstractions in SystemC

The SystemC elements discussed above allow modeling of electronic systems at multiple abstraction levels. The Figure 3.4 shows two such levels, where communications between two hardware components have been modeled using detailed and functional levels.

SystemC provides a channel class known as `sc_signal`, which can be used to model a single electronic signal or a set of signals representing a bus. Figure 3.4(a) shows the detailed modeling approach where each of the data, address and control signal has been modeled. We can see how ports, interfaces and channels are connected. The ports can be used for input (`sc_in`), output (`sc_out`) or both (`sc_inout`) and channels implement appropriate interfaces *e.g.* `sc_signal_in_if<>`, `sc_signal_out_if<>` or `sc_signal_inout_if<>` of the required size. To ensure proper signal transfers, the connected hardware models

have to implement a precise communication protocol, usually in terms of a *Finite State Machines* (**FSMs**).

Figure 3.4(b) shows the abstract modeling approach where the detailed signal level communications have been replaced by equivalent but higher level function calls. Signals are not used and the functionality of channels is moved inside the slave device models, which implement a specific interface, such as `if_read_write` in this example. The implemented methods *e.g.* `read()` and `write()` are made accessible to other modules using `sc_export` type ports. Master devices connect to the slave devices using the `sc_port` type ports and directly use the required functionality. This particular modeling style is known as *Transaction Level Modeling* (**TLM**) and results in much faster simulation platforms than the ones modeled at signal level.



(a) Detailed Communication Modeling (b) Functional Communication Modeling

Figure 3.4: Communication Modeling Abstractions in SystemC

3.4 Abstraction Levels and Simulation Models

The use of multiple abstraction levels is indispensable for the design of complex systems, such as MPSoC, and allows us to gradually refine a given specification to the final implementation. The use of software programming languages for the purpose of hardware modeling was a key factor in the genesis of these different abstraction levels [JBP06]. Each abstraction level has an associated simulation model, which enables the validation of full or a part of the software stack, with a certain level of architectural and timing accuracy. More recently, the semi-conductor industry has moved towards the standardization of languages, which support such abstraction levels *i.e.* [SYS] and [Ber06]

Simulation techniques are now widely in use by the actual SoC design teams [Ghe06] and hardware/software co-simulation techniques have emerged as a viable and mature *Computer-Aided Design* (**CAD**) technology. Using different abstraction levels, the designers of embedded systems can perform *precision vs. performance* trade-offs, depending on the development stage and many texts such as [MEW02, SLM06] have addressed these issues in detail. Figure 3.5 shows the principle abstraction levels, as proposed in literature, as well as the corresponding

simulation models (if applicable). In certain cases, hardware/software interactions are also modeled, using a hybrid interface model. In the following sections, we briefly introduce these abstractions levels, along with key benefits and shortcomings.

3.4.1 System Level

The *System Level* (**SL**) as shown in Figure 3.5 ❶ is the most abstract level and refers to the executable specifications of the application. All tasks of the application communicate using abstract channels and there is no difference between hardware and software tasks. Such a simulation model could be implemented using a specific language such as Simulink [PJ07, HHP+07], SystemC or it may even be realized using C/C++ [VKS00]. Simulation models at this level offer high performance but are purely functional, as they contain neither platform specific details nor timing information. The software tasks have no relation to the final software stack and it is not yet decided as which of these tasks will be implemented in hardware. Nevertheless, this level is used to verify certain functional properties of the application and can serve as a reference model.

3.4.2 Cycle Accurate Level

On the other extreme, we consider the *Cycle Accurate* (**CA**) level shown in Figure 3.5 ❺, as it offers the most precise modeling and simulation of hardware and software components *w.r.t.* to the real hardware platform, that we consider in this thesis. At this level, the complete software stack is cross-compiled for the target platform and the final binary is loaded into the simulated memory during platform initialization. The simulation platform models all of the hardware components with cycle accuracy and the interconnection components use the detailed communication modeling approach, as shown in Figure 3.4(a) and all of the signals are represented exactly as they are in a real platform. As all the functional parts of the platform are modeled using hardware components, a hybrid hardware/software interface model is not required. Moreover, as hardware components are modeled using cycle accuracy, their simulation time advances on each clock cycle, consequently this abstraction level does not require timing annotation of component models. This level is usually the last step in the design flow, followed by the modeling of system at RTL level, in order the synthesize the final circuit.

The target CPU is modeled by an *Instruction Set Simulator* (**ISS**), which simulates the internal details of processor operations, including pipeline stages, instruction dependencies, delay slots *etc.* using an infinite execution loop. During each iteration of this loop, the processor fetches the next instruction from the instruction cache, using the current value of program counter. The fetched instruction is then decoded and desired operations are performed on the simulated registers and memory, including load/store, arithmetic and logical operations. The memory write-back stage stores the resultant values, if any, into the simulated data cache. In case of an instruction or data cache miss, the corresponding cache-line is searched in the simulated memory using the cycle accurate interconnection models, which implement the same communication protocol as in real hardware. Memory hierarchies are also represented in an accurate fashion, such as cache-coherence in multi-processor systems, enabling the modeling of software execution timings in a precise manner.

On the down side, this level suffers from very low simulation performance, as the ISS decodes all the instructions, even if they are executed multiple times, and as a side effect of

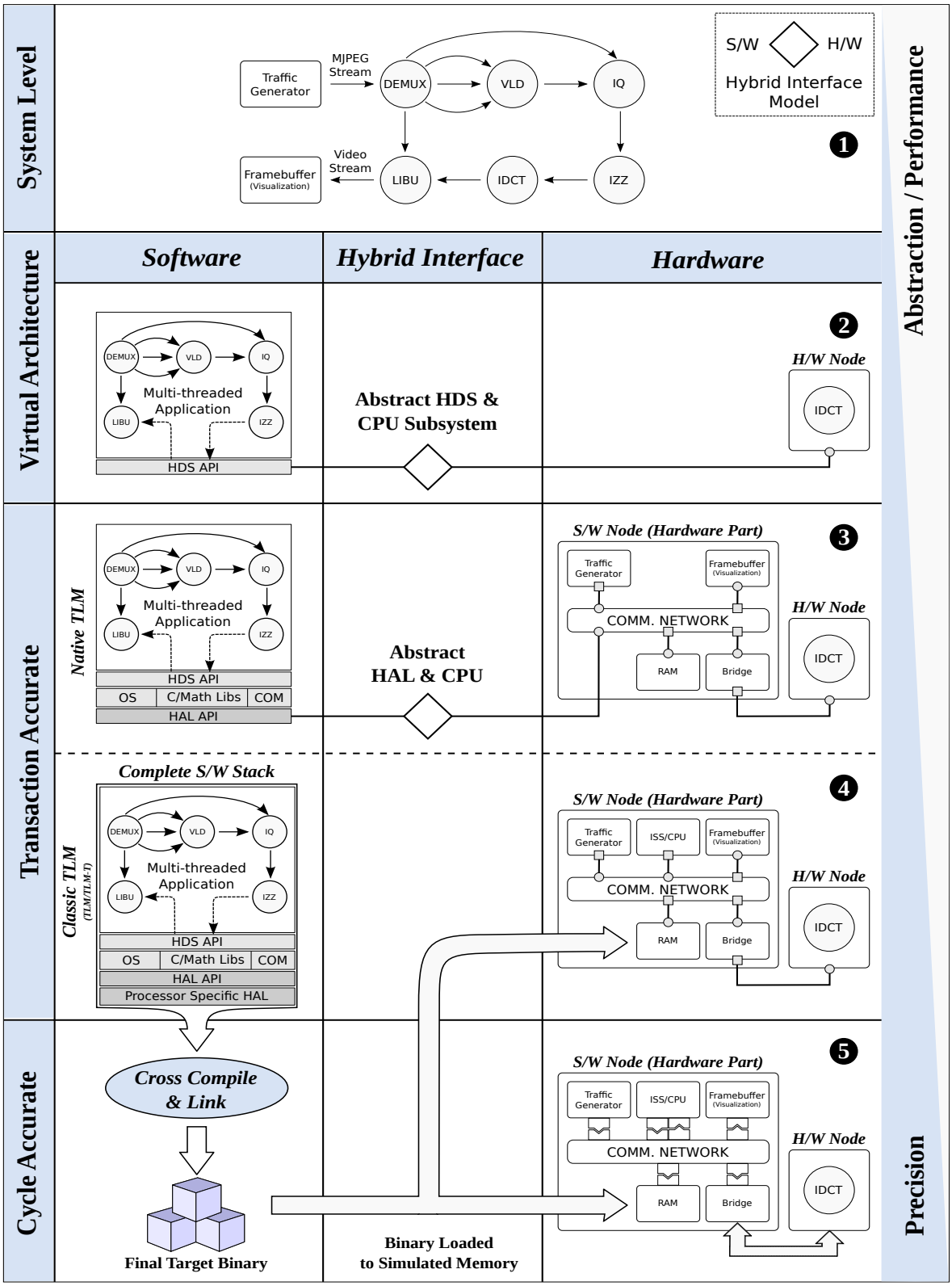


Figure 3.5: Abstraction Levels and Simulation Models

detailed component/communication modeling. However, this level is suitable for functional validation of hardware components, such as their FSMs, low-level software implementations and even synchronization mechanisms in multiprocessor systems that require precise time modeling.

As the cycle accurate and system level models lie on two extremes, many researchers have proposed intermediate abstractions including [CG03, Don04, vdWdKH+04, PGR+08], offering a compromise between precision and performance. Following sections briefly discuss these intermediate levels.

3.4.3 Virtual Architecture Level

The first refinement step *i.e.* *Virtual Architecture* (**VA**), partitions the hardware and software tasks and defines a hybrid interface layer between them, as shown in Figure 3.5 ❷. The hybrid interface layer is an abstract implementation of *Hardware Dependent Software* (**HDS**) and CPU subsystem providing operating system primitives, as well as explicit communication components using abstract channels. These communication components specify an end-to-end communication path, which could be useful in providing statistics about communication requirements. The software is only dependent on the HDS API layer, further refinements do not require modifications to the application code. As a result software application code can be validated using this abstraction level.

3.4.4 Transaction Accurate Level

The next abstraction level is known as *Transaction Accurate* (**TA**) or more familiarly *Transaction Level Modeling* (**TLM**), primarily focuses on communication abstractions to gain performance benefits. The TA models operate in terms of transactions rather than signals, as shown in Figure 3.4(b). Two main approaches have been proposed at this level, based on the techniques used for simulating the software stack. The ISS-based platforms known as Classic TLM models that benefit from abstractions in communication modeling only and use target ISSes for interpreting instructions, as in cycle accurate models. The second approach, known as Native TLM models, replaces the ISS based processors with abstract CPUs and HAL implementations, in addition to the transaction level communication modeling, and profit from performance benefits in both computations and communications.

3.4.4.1 Classic Transaction Accurate Level

The Classic TLM level is very similar to the CA level and is shown in Figure 3.5 ❹. At this level, the signal-based communication interfaces in hardware component models are replaced with transaction level function calls. As a result, the processing elements *i.e.* the ISSes also use transactional interfaces while keeping the internal implementations similar to CA level. The complete software stack remains unmodified *w.r.t.* to the CA level and is still cross-compiled for target machine. The final target binary is loaded into the simulated memory and interpreted by the ISSes.

Regardless of how precisely the component models and system architecture has been defined, the overall simulation lacks precision. The principle inaccuracy comes from the TLM-based communication modeling that does not advance the simulated time *i.e.* all transactions take *zero-time*. Such simulation architectures that lack time modeling in communication interfaces are also known as *Programmer's View* (**PV**) models. The communication interfaces

can be time-annotated to improve the simulation accuracy and such models are commonly referred as *Programmer's View with Time* (**PVT**) [CMMC08] or *Transaction Level Modeling with Time* (**TLM-T**) in literature [TAB07, HSAG10]. More recently, parallel simulation platforms targeting multi-core machines have appeared in [MMGP10, PMGP10], which refer to such interfaces as *Transaction Level Modeling with Distributed Time* (**TLM-DT**).

The principle deficiency in such models comes from the use of ISSes, which consume significant processing time as they interpret instructions and large number of transactions (instruction fetching and data accesses) that are issued on the communication network. Due to these limitations, the validation of large software stacks such as image processing applications running on top of an OS becomes impractical.

3.4.4.2 Native Transaction Accurate Level

The concept of native execution is well known to embedded system designers, where they usually compile a sub-set of software stack for the host machine instead of the target embedded system. The key benefit is faster execution at the price of a more abstract view of the execution environment.

Embedded software is organized in layers, where each layer provides a set of APIs to the upper layers and hides lower level implementation details. In lower software layers, hardware dependence increases, making them less portable and thus decreasing their possible validation using native execution. For example, an application which relies on the *Portable Operating System Interface* (**POSIX**) standard, can be compiled on a native machine running a Linux operating system as well as on an embedded system, which provides an equivalent API.

The key idea in recent native simulation [BBY+05] works lie in confining the hardware specific details to the HAL and providing a target independent HAL-API to operating system and libraries. An abstract HAL and CPU implementation provides the interface between software and hardware models, as shown in Figure 3.5 ❸. As the hardware models do not use an ISS-based CPU, timing annotations are directly inserted into the software stack [GHP09] running on the abstract CPU implementation.

Compared to the classic TLM simulation, which uses ISSes as processing elements, native simulation is much faster but less precise as it is strongly dependent on the annotation accuracy for modeling of software execution timings. Native simulation, as of today, does not come without its problems including how it resolves conflicts in the address-spaces of simulated software and hardware platform and simulating software stacks that use *Memory Management Unit* (**MMU**).

3.5 Native Transaction Accurate Simulation

The Native Transaction Accurate simulation platform proposed in [Ger09] is derived from [BBY+05] and generic TLM techniques. The key aspect in this methodology is to organize the software stack in layers, by providing well-defined interface layer APIs. This layered organization improves software portability and allows for the validation of most of the software stack. Only the hardware specific parts, which are encapsulated in the HAL layer, cannot be validated using this approach as they will *eventually* be replaced with target processor specific implementation (usually in target assembly) in the final software stack.

3.5.1 Hardware Abstraction Layer

Native execution of software is impossible if the interface between software and the underlying hardware *i.e.* processor and platform, is not clearly defined. In essence the hardware specific details should be confined to a single layer so that the rest of software can be compiled for native machine. The lowest possible software API layer that could be used to provide such portability, is commonly referred as *Hardware Abstraction Layer* (**HAL**)[2]. The concept of HAL in SoC domain was introduced by [YJ03], which indicate the provision of low-level functionalities including platform startup routines, context switching, cache and virtual memory configurations, synchronization and all of the fine details for hardware management.

In the native TA simulation platform, HAL declarations and implementation are segregated. The software stack contains *external* HAL function declarations and the hardware platform provides the actual HAL API implementation. Native software is compiled as a dynamic library and provides definitions of all symbols except for HAL API calls. Dynamic library manipulation facilities of the host machine are used to load the software stack into the simulated memory and resolve the external HAL references. This scheme enables portability, as the same software stack can be recompiled for a different hardware platform that provides the same HAL interface, including the real hardware platform. In addition, it enables the hardware platform to intercept all the HAL API calls and redirect them to appropriate platform components.

Listing 3.1 shows how a HAL function is declared in the software stack and Listing 3.2 provides an example implementation of the same HAL function by the native simulation platform. We can observe that the implementation uses *load linked* and *store conditional* functions provided by the hardware platform to ensure proper semantics of the *test_and_set* synchronization primitive. The `m_cpu_id` is taken from the current processor instance and passed to load linked and store conditional operations for ensuring atomicity in multiprocessor environments.

Listing 3.1 External Declaration of HAL API for `CPU_TEST_AND_SET()`

```
1 #ifndef __CPU_SYNC_H__
2 #define __CPU_SYNC_H__
3
4 extern long int CPU_TEST_AND_SET (volatile long int * spinlock);
5
6 #endif /* __CPU_SYNC_H__ */
```

Listing 3.3 gives an implementation of the same HAL API for ARMv6 processor based platform, which can be used to generate cross-compiled software binaries. Such binaries can be potentially used on a real ARMv6 based platform as well on an ISS-based TA or CA simulation platform, as shown in Figure 3.5 ❹ and ❺, respectively.

3.5.2 Native Software Execution

Abstract HAL and CPU models form the hybrid interface layer in the native TA abstraction level, as shown in Figure 3.5 ❸. *Execution Units* (**EUs**), as they are referred in [BBY+05, Ger09],

[2] HAL is an abstraction layer, implemented in software and hides hardware architecture specific details from the upper software layers that execute on top of it. Many types of HALs exist, where each operating system gives its own definition and version of APIs, with more or less abstract descriptions.

Listing 3.2 HAL Implementation of `CPU_TEST_AND_SET()` for Native Processor

```
1  long int CPU_TEST_AND_SET(volatile long int * spinlock)
2  {
3      long int ret;
4
5      do {
6          if((ret = p_io->load_linked((uint32_t*)spinlock, m_cpu_id)) != 0)
7              break;
8      } while((ret = p_io->store_cond((uint32_t*)spinlock, 1, m_cpu_id)) != 0);
9
10     return ret;
11 }
```

Listing 3.3 HAL Function `CPU_TEST_AND_SET()` for ARMv6 Processor

```
1  long int CPU_TEST_AND_SET (volatile long int * spinlock)
2  {
3      register long int ret, temp = 1;
4
5      __asm__ volatile("ldrex   %0, %3\n"
6                       "cmp     %0, #0\n"
7                       "strexeq %0, %2, %1\n"
8                       : "=&r" (ret), "=m" (*spinlock)
9                       : "r" (temp), "m" (*spinlock)
10                      : "memory"
11                     );
12     return ret;
13 }
```

are used to model this hybrid interface, by providing an implementation of HAL layer, as well as hardware threads (SystemC Threads) to model the processing elements. Figure 3.6 shows the detailed organization of HAL and CPU threads using the abstraction of EUs, as well as how software executes on top of them. The dashed arrows indicate how the control flows between hardware models and software components.

The software stack is compiled as a dynamic library, with external HAL API references, and its execution is handled by one or more EUs. A C/C++ HAL interface layer, Figure 3.6 Ⓑ , bridges the gap between the HAL calls and their implementation in EUs. The software execution context as provided by a processor, is modeled using a SystemC thread shown as Figure 3.6 Ⓐ. Multiple EUs can be used to model an MPSoC system where hardware concurrency is provided by SystemC simulation kernel using SystemC threads. The execution of software on an EU is very similar to a real CPU and following key steps are involved in this process.

❶ At the start of simulation, each EU creates a thread for software execution, which calls the `reset()` function implemented in HAL. This `reset()` function performs low level initializations, including interrupt handlers, stack *etc.*, before passing control to the software entry point.

❷ Once the low level initializations are complete, the `reset()` function calls the software entry point, usually the bootstrap code of operating system. The OS bootstrap initializes services including memory and process management, file systems, device drivers *etc.*, before passing control to the software application.

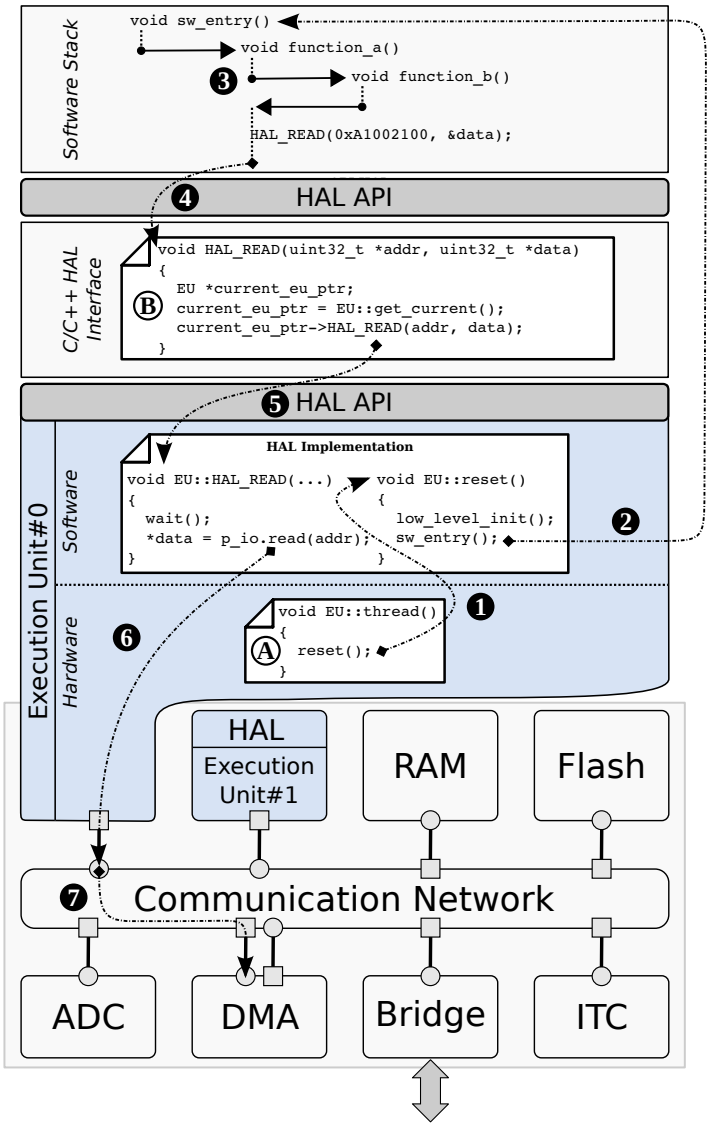


Figure 3.6: Principle of Native Execution on a Transaction Accurate Platform using HAL Interface and Abstract CPU Models [Ger09]

❸ The software executes in a sequential and continuous fashion, without interacting with the simulation platform *i.e.* software execution takes *zero-time w.r.t.* the simulated time in the hardware platform. The software continuous to execute until a hardware specific service is required *e.g.* read request for a specific device register, resulting in a HAL API call.

❹ When the software is executing, the hardware platform completely loses control over it until a HAL API call is invoked. Even though the EUs provide the HAL implementation, they do not directly intercept HAL calls and a C/C++ interface layer Ⓑ is introduced for the following reasons.

➠ The interface layer is required when the software and hardware components use different programming languages, such as C and C++ for software and hardware

implementations, respectively.

- The executing software does not have the *knowledge* of the underlying platform, whether it is a real processor or an *Execution Unit* (**EU**) and the interface layer bridges this gap by redirecting the HAL calls to the appropriate EUs.

❺ Once a HAL function in an EU executes, it gives control back to the SystemC simulation kernel by calling the `wait()` function. The simulation kernel can then schedule other processes running in parallel, as well as advance the simulated time. Meanwhile, the calling process may be suspended for fixed time interval or until a future event takes place.

❻ The HAL function forwards the request to the appropriate I/O port of the EU, as it happens on a TLM based platform, where each such request translates to a read or write function call that is invoked for the communication component.

❼ Finally the address decoder of the communication component decodes the I/O address and forwards the request to the appropriate device model. Once the request has been satisfied, the function call stack unwinds and software continues to execute sequentially until another HAL API call is executed.

This approach enables the validation of a software stack, either with or without an OS that is dependent on the HAL interface only. The capability to model multiple processing elements is independent of the HAL abstraction. If the simulation platform is capable of identifying these processing elements and provide synchronization mechanisms between them, then the validation of an SMP operating system is possible, at the cost of porting the OS for the given HAL.

3.5.3 Target *vs.* Host Address Spaces

Native simulation platforms face two types of dependencies, mainly resulting from the native compilation of software. The differences in *Instruction Set Architecture* (**ISA**) of host and target processors, as well as specific details of hardware modules, need to be addressed in the first place. These differences are resolved by explicitly using the HAL API for *all* interactions between software and hardware components, resulting in a *hardware-independent* software stack, except for the HAL implementation. Second source of dependencies arise due to the memory representations in hardware and software components. To be precise, two different and possibly conflicting address-spaces have to be considered, namely target (Ⓣ) and host (Ⓗ) address-spaces, as shown in Figure 3.7.

Ⓣ The hardware platform, although compiled on the host machine, *simulates* the target addresses that are known at compile time *i.e.* the hardware components address mappings have been defined in advance by the system designer and the platform address decoders use these mappings for communication between the hardware models.

Ⓗ The software stack is compiled for the host machine and all memory references are unknown at compile-time. The software addresses become known at runtime when the software library is *actually* loaded into the host memory. Usually its the SystemC process, which loads the software library and provides the execution context.

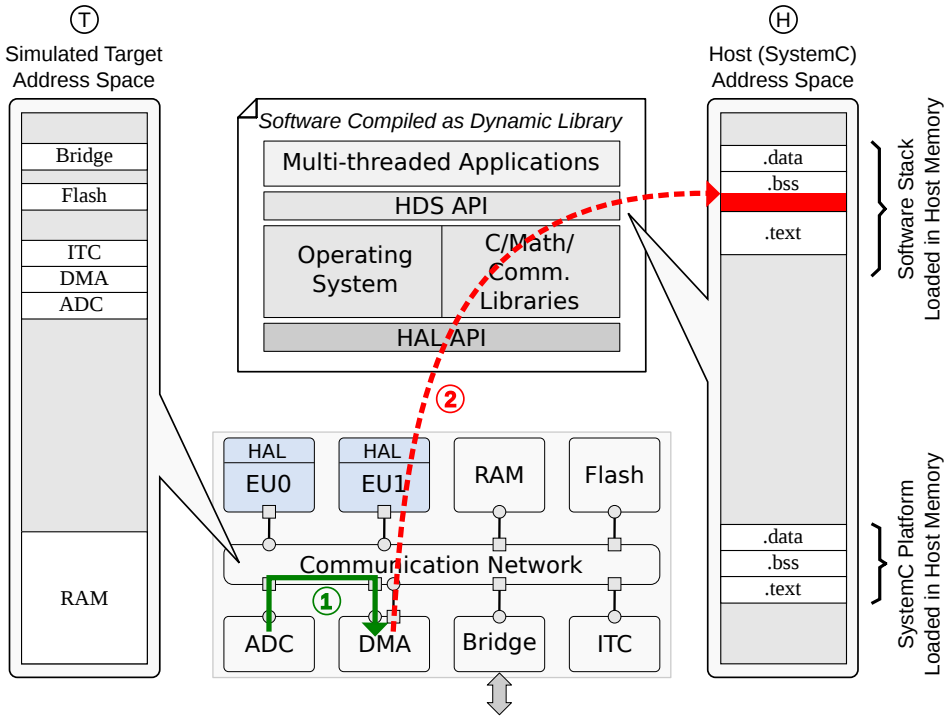


Figure 3.7: Target *vs*. Host Memory Representations. (Adapted from [Ger09])

The *target vs. host* address-spaces issue does not appear in ISS based simulation platforms, as both hardware and software components view the *same* address-space *i.e.* the target address-space. The differences in address-spaces, make it impossible to model certain interactions between hardware and software components.

Listing 3.4 A software thread processing data input from ADC using DMA [Ger09]

```
1  #define ADC_RCV_REG 0xA0001004  /* In Target Address Space */
2
3  static int  data_in[1024];      /* In Host(SystemC) Address Space */
4  static int data_out[1024];      /* In Host(SystemC) Address Space */
5
6  void thread_compute()
7  {
8      /* Local Variables, if any */
9      while(1)
10     {
11         /* Initiate the DMA Transfer */
12         DMA_TRANSFER(ADC_RCV_REG, data_in, 0x0, 0x4, 1024);
13         for(...)
14         {
15             /* Do computations in parallel to DMA transfer */
16             data_out[i] = compute(data_in[i]);
17         }
18     }
19 }
```

We illustrate this problem using a DMA based example, which is commonly found in MPSoC systems. Listing 3.4 shows a software thread, which performs a certain computation on the data input from an *Analog to Digital Converter* (**ADC**) component. In order to process

Listing 3.5 DMA initialization function for data transfer [Ger09]

```
1  #define DMA_BASE_ADDR    0xB0000000
2  #define DMA_CTL_REG      DMA_BASE_ADDR
3  #define DMA_SRC_REG      DMA_BASE_ADDR + 0x04
4  #define DMA_DST_REG      DMA_BASE_ADDR + 0x08
5  #define DMA_SRC_INC      DMA_BASE_ADDR + 0x0C
6  #define DMA_DST_INC      DMA_BASE_ADDR + 0x10
7
8  void DMA_TRANSFER(void *src, void *dst, int src_inc, int dst_inc, int size)
9  {
10     HAL_WRITE_UINT32(DMA_SRC_REG, src);       /* Write Source Address */
11     HAL_WRITE_UINT32(DMA_SRC_INC, src_inc);   /* Write Source Incremement */
12     HAL_WRITE_UINT32(DMA_DST_REG, dst);       /* Write Destination Address */
13     HAL_WRITE_UINT32(DMA_DST_INC, dst_inc);   /* Write Destination Increment */
14     HAL_WRITE_UINT32(DMA_CTL_REG, size);      /* Write Size & Start Transfer */
15     return;
16 }
```

the input data, the software thread needs to copy of this data in a software buffer, declared as a local array `data_in[]`. It initiates a `DMA_TRANSFER` and starts to process the input data, while the data is being copied in parallel by the DMA component.

Listing 3.5 shows the DMA initialization function, which setups the source and destination addresses, as well as source and destination increments and data transfer size. Writing the transfer size to control register of DMA, at Line 14, starts the actual data transfer. The DMA needs to access two different address-spaces to perform this transfer:

① The source address, *hard-coded* as `0xA0001004` in software, lies in the simulated target Ⓣ address-space. The communication component knows this address from the predefined address mappings, so it can forward requests from DMA to the appropriate component *i.e.* ADC in this case, without any problems.

② The destination address *i.e.* the memory address assigned to software buffer `data_in[]`, is resolved dynamically when the binary is loaded in the host (SystemC) address-space Ⓗ. As the communication component does not know this address because it lies in the host Ⓗ address-space, it cannot fulfill the write requests from DMA.

The DMA example highlights the limitations of the TA level native platforms, when a hardware component wishes to access the memory seen by the natively executing software. Without modifying the hardware communication mechanism or software code, such interactions cannot be supported in native simulation. As discussed earlier, most of the real architectures contain components like DMA or similar master devices, apart from processors, which require such interactions.

Similarly, the inverse case also poses problems when the software tries to *directly* access a hard-coded hardware address. Listing 3.6 shows an example where the software tries to access the video memory using a pointer initialized with a hard-coded memory address. On Line 13 of the `vmem_clear_screen()` function, a write to the buffer pointer would result in a segmentation fault. This type of memory accesses do not cause problems in real or cycle accurate platforms, whereas in native simulation such accesses are impossible as the software application cannot see the memory regions modeled by the hardware components. In order to tackle this problem, all accesses using hard-coded addresses are usually *forbidden* or an intermediate API is used for accessing such memory zones.

Listing 3.6 A simplified implementation of a Video Memory Clear Screen function

```
1  #define VMEM_BASE_ADDR 0xC4000000
2  #define VMEM_WIDTH     256
3  #define VMEM_HEIGHT    144
4  #define VMEM_SIZE      (4 * VMEM_WIDTH * VMEM_HEIGHT) /* 4 times for RGB + Alpha */
5  #define VMEM_LIMIT     (VMEM_BASE_ADDR + VMEM_SIZE)
6
7  void vmem_clear_screen()
8  {
9      volatile unsigned int *vmem_ptr = (unsigned int *) VMEM_BASE;
10
11     while(vmem_ptr < VMEM_LIMIT)
12     {
13         *vmem_ptr = 0x00000000;                    /* Write to Video Memory */
14         vmem_ptr++;
15     }
16 }
```

3.5.4 Using a Unified Address Space

A key solution to the *target vs. host* address-spaces issue, was presented in [Ger09]. This solution advocates the use of a uniform address-space for resolving the address-space conflicts and proposes a number of modifications to both hardware and software components. We briefly describe these modifications in this section using Figure 3.8, which shows the key components of this solution.

As this solution unifies the target address-space into the host address-space, it necessitates changes to the hardware platform components, as its first requirement, which is usually *un-acceptable* to most of the hardware IP users. The host address-space and natively compiled software have a dynamic character, as the software addresses become known at runtime. Instead of using predefined address mappings, the solution proposes the use of dynamically allocated host addresses in SystemC IP models.

As a second requirement, each slave component in the hardware platform provides information about its *Mappings* Ⓐ and *Symbols* Ⓑ that it defines. A mapping defines the memory regions and device registers used by a slave component, usually in terms of base address, size and name. Symbols are defined as name and value pairs, which need to be resolved in the software application. An operating system specific and non SystemC component is introduced, known as *Dynamic Linker*, which inquires each of the hardware components Ⓒ, through the communication component, about their mappings and symbols, and patches them Ⓓ in software stack during platform initialization. The communication component is also modified; instead of using statically known target addresses, it now uses the dynamic mappings for constructing its address decoding table Ⓔ. This decoding table contains all the memory regions of the simulation platform that are accessible using the interconnection component.

The software is also restricted, in the sense that hard-coded addresses cannot be used, as to avoid segmentation faults. Instead the hardware specific address resolution is delayed until the start of simulation, using external pointer declarations. Similarly the link time symbols, usually defined in linker scripts, are also restricted and cannot be declared as constants. Linker symbols are declared as pointers using a C/C++ source file and resolved during platform initialization, by the dynamic linker.

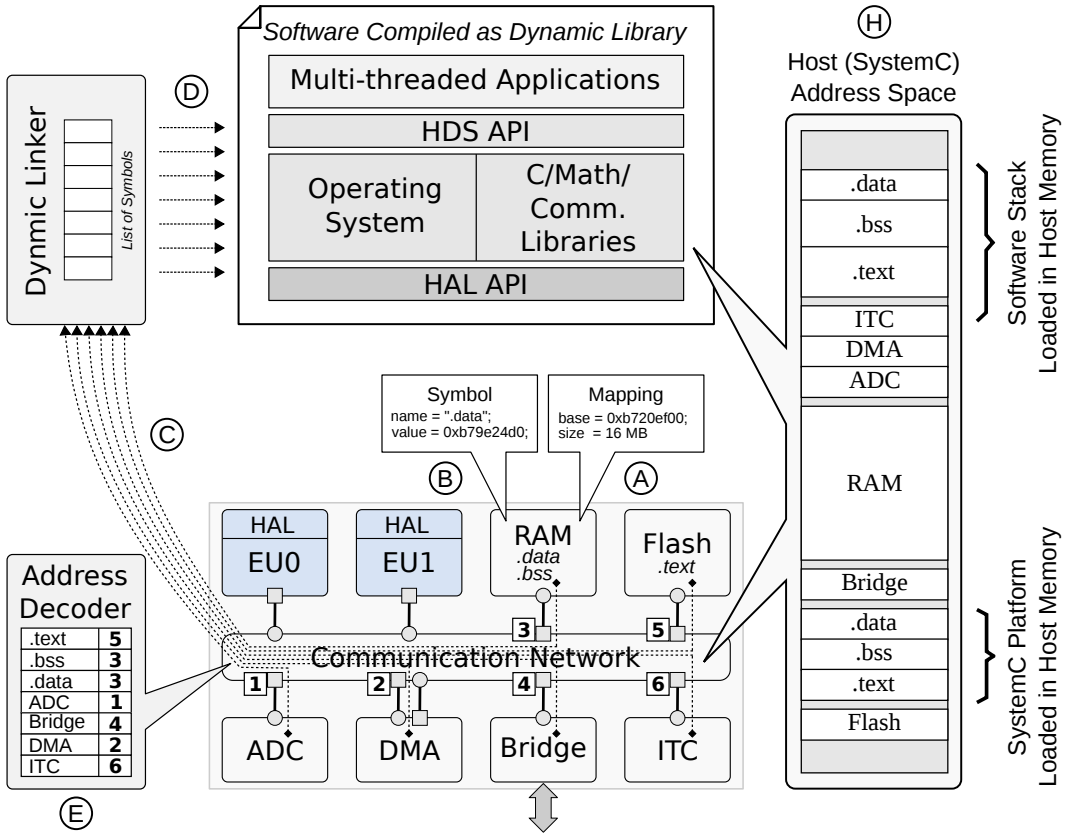


Figure 3.8: Native Uniform Memory Representation

3.5.4.1 Limitations of Unified Address Space

In the preceding section, we described the changes required for creating a unified address-space. We list the key limitations of this approach in the following text:

- Each platform component, including the communication network, if it occupies a memory zone must:
  - allocate memory zones that are potentially accessible from software, respecting the real memory mappings of the component.
  - allow access to the descriptions of allocated memory zones.
- Each platform component must provide a list of symbols that are expected by software, using a *(name, value)* pair and allow access to its symbols list.
- The communication network must build its decoding table by using descriptions of allocated memory zones, once the hardware platform has been initialized. These zone descriptions are provided by the peripheral devices and are accessible through the input/output ports of these components.
- The memory components of the platform containing software segments (.bss, .data, .text *etc.*), must make them accessible through the communication network. All memory

components must also provide descriptions of the allocated memory zones and these are necessary for building the address decoding table of the communication network.

- The use of hard-coded addresses in software is not supported, as the addresses of peripheral devices are unknown at compile time. Moreover, using such addresses on the host machines generate segmentation faults, as these addresses are usually invalid in the SystemC process mapping.

- Linker symbols cannot be directly defined during the software linking phase because it is impossible to modify their addresses during execution, as required by the dynamic linking phase. Instead pointers must be used in software, including the device drivers and operating system, for defining physical memory addresses.

- Execution of complex multi-tasking operating systems such as Linux, which use *Memory Management Unit* (**MMU**) for virtual memory management, are not supported using the unified address-space approach.

3.5.5 Software Performance Estimation

The key strengths of native techniques are the improved simulation performance and ease of functional validation. On the downside, native techniques suffer from the absence of time modeling in software, making them inherently inaccurate.

Native software executes in the context of hardware threads such as *Execution Units* (**EUs**), and renders control to the hardware process *e.g.* SystemC kernel, only when a HAL function is executed. In between two HAL function calls, the native software executes in *zero-time* and the hardware process has no control over it. This is a consequence of cooperative multi-tasking approach in SystemC, as discussed in Section 3.3.1. To enable time modeling, the only feasible solution is to insert performance annotations in software that could render control back to the hardware process.

Annotations can be inserted at source level [GMH01, KKW+06, MSVSL08] or at an intermediate level [WH09] using a compiler's *Intermediate Representation* (**IR**). Quite a few automated software annotation techniques have been proposed in literature [LP02, CGG04, CHB09b, GHP09]. The IR, usually a kind of bytecode, contains all the semantic information and is used throughout the compilation process including optimization phases, as shown Figure 3.9. The idea is to run the retargetable compiler, such as *Low Level Virtual Machine* (**LLVM**), as if the code will be generated for the target processor, including target-specific optimizations, but instead of emitting assembly for the target machine Ⓧ, either emit it for the host Ⓨ or generate C and compile it for the host Ⓩ.

An annotation pass analyzes the target code and inserts calls to an annotation function at the beginning of each *basic block*[3] in the target aware IR. The annotated IR is then used to generate a native binary object, which has an equivalent *Control Flow Graph* (**CFG**) to that of the target object and can be simulated on the host machine. The values to be added as annotations depend on the processor architecture and in addition on the platform, for the ones that make external accesses.

[3] A basic block is a set of consecutive instructions having exactly one entry and one exit point. At runtime, if the first instruction in a basic block is executed, the rest of instructions must also be executed in exactly the same order as they appear, before control is transfered to any other basic block.

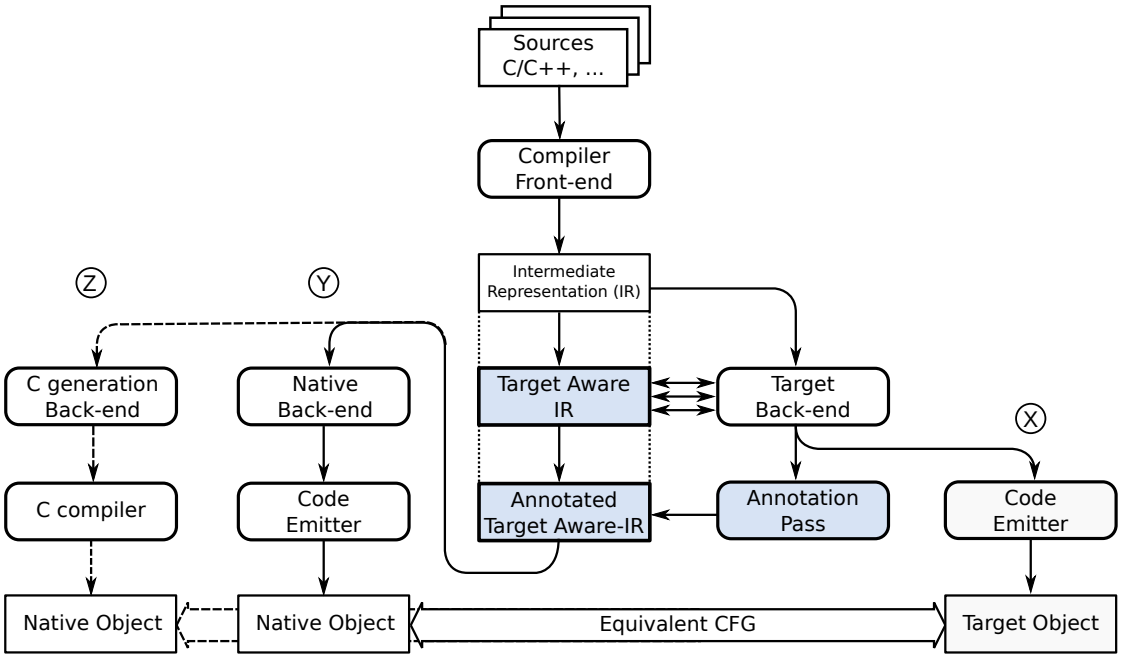


Figure 3.9: Native Software Compilation with Target Specific Annotations [PFG+10]

Execution of annotation calls provide the hardware platform (SystemC) with an opportunity to advance the simulated time using `wait()` function calls, enabling software performance estimation. The annotation techniques should consider all types of memory accesses, including cache effects and software execution itself *i.e.* accesses to the program memory. Reflecting memory access timings in native software execution is a difficult problem, as all memory accesses take place natively *i.e.* the underlying platform is unaware of such memory accesses.

One of the objectives of this work, is to propose a native simulation technique that could decrease the number of limitations discussed in Section 3.5.4.1, while keeping support for software performance estimation.

The above strategy works well for simple RISC processors but is difficult to apply for complex *Very Long Instruction Word* (**VLIW**) architectures *e.g. Texas Instruments* (**TI**) C6x and Nexperia PNX series processors. Following list gives a few factors limiting the applicability of this approach to VLIW machines:

- A VLIW specific backend is required in the retargetable compilation frameworks, such as in LLVM, in order to generate target specific code.
- The retargetable compiler is able to produce VLIW binary code, which is comparable to a standard vendor-supplied VLIW compiler in terms of optimizations.
- A VLIW specific annotation pass is implemented, which takes performance measures from parallel VLIW code and annotates them to serial IR code, to build a VLIW-aware but VLIW-independent IR.
- Source code for the VLIW software binaries is available.

In the absence of a retargetable compilation framework that supports a VLIW architecture, native simulation degrades to functional validation, as target specific annotations are impossible. For example, LLVM compilation infrastructure does not support VLIW machines, as of today. Even if such a compiler is available, the target specific optimizations are much less efficient as compared to the vendor-supplied toolchains that are usually capable of generating highly optimized VLIW binaries. This aspect further limits the generation of native binaries that have an equivalent CFG *w.r.t.* the cross-compiled VLIW binaries.

Implementation of a VLIW specific annotation pass is difficult, as the compiler needs to keep correspondence between target independent and target specific IRs. Such correspondence would be hard to establish, as optimization passes applied by the VLIW backend would be much different from the standard IR optimizations for non-VLIW machines. This limitation emerges from the architectural differences between parallel and scalar architectures.

Availability of software sources is also important, especially when the code has been optimized for a certain architecture. Quite often, we find ourselves in situations when the software vendors provide *closed* source libraries only, rendering performance estimation using annotation techniques impossible.

Considering the above limitations, it is interesting to see if generation of native executables is possible using the static translation[4] principle, from the cross-compiled VLIW binaries. We emphasize on static translation, in order to avoid any run-time translation overheads and to generate optimized native binaries.

In the following text, we review fundamental features of VLIW machines and discuss key problems during translation from parallel VLIW instructions to serial native code. We take the example of TI C6x series processors as they exhibit some of the most problematic features found in VLIW machines.

3.6 VLIW Processor Architecture Simulation

The VLIW processors can issue and complete more than one operation at a time, commonly referred as *Instruction Level Parallelism* (**ILP**), which is the key to their increased performance and justifies their use in broad range of applications. The VLIW software compiler has the responsibility of encoding this concurrency information in the generated instructions. This explicit encoding of parallelism, also known as *static scheduling*, results in reduced hardware complexity, which makes VLIWs good candidates for demanding embedded devices. The generic architecture of a VLIW processor is shown in Figure 3.10.

3.6.1 Modeling Parallelism and VLIW Pipelines

We demonstrate the difficulties associated with modeling of a VLIW architecture on a desktop machine, using an example shown in Listing 3.7 that uses TI C6x processor instruction set. On C6x processors, multiple RISC-like instructions are bundled together and execute in parallel, in a single CPU cycle. These are known as *Execute Packets* and can contain upto eight instructions. Listing 3.7 contains 10 instructions grouped into 5 execute packets, starting at instructions 1, 5, 7, 9 and 10. The || symbol in front of an instruction specifies parallelism,

[4]Static translation refers to the off-line processing of input binaries, similar to compilation but partially in the opposite direction, and does not require run-time translation support as in *Dynamic Binary Translation* (**DBT**).

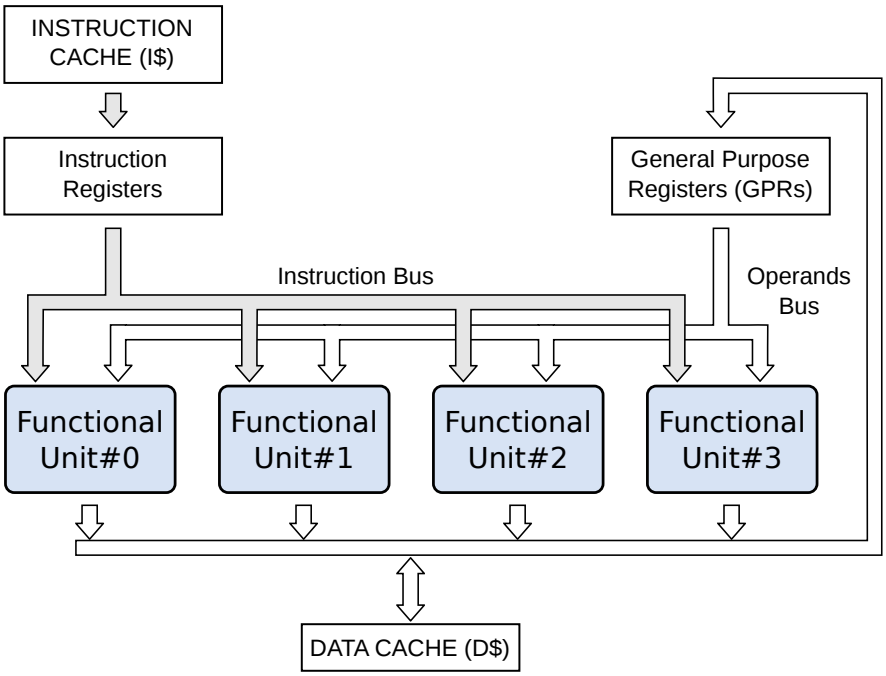


Figure 3.10: Generic Architecture of a VLIW Processor

such as before Line 2 through 4 indicating that these instructions belong to the first execute packet, and will be executed in parallel to the ADD instruction at Line 1.

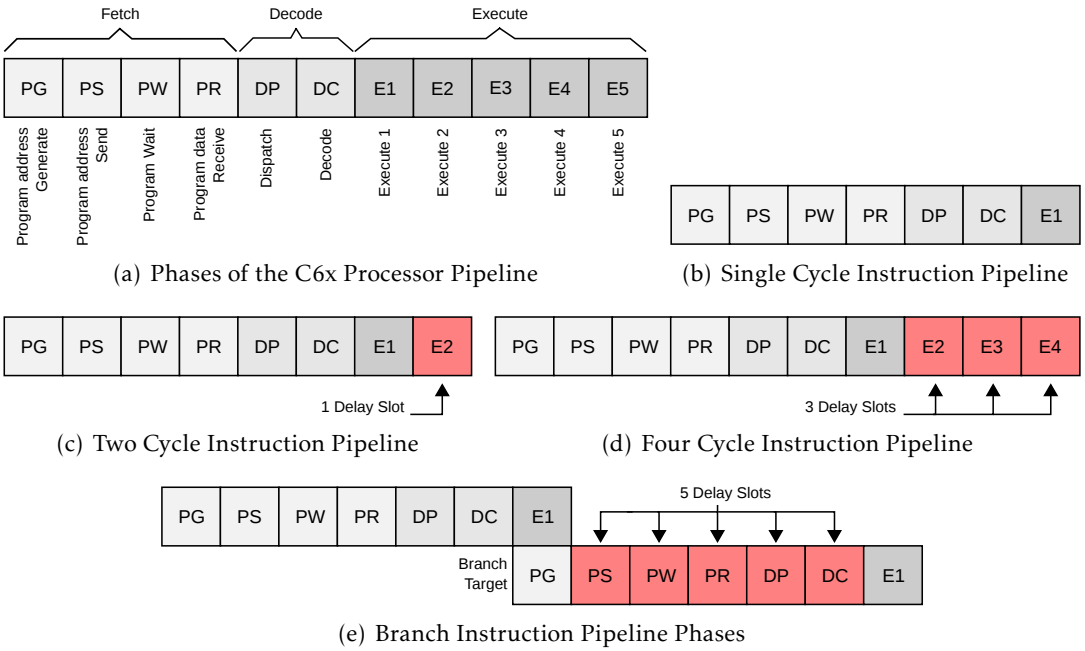
Listing 3.7 An Example VLIW Code (TI C6x)

```
 1      [!B1] ADD.L1     B1,4,B3         ; DSP Cycle 0
 2   ||       STW.D1     B6,*B5++[1]
 3   ||        MV.S1     B3,A4
 4   || [B0]    B.S2     0x010100        ; 1
 5         MPYSU.M1     B6,A4,B6
 6   ||     MPYU.M2     A4,B4,A3        ; 2
 7           SUB.L1     1,B0,B0
 8   ||      SHR.S1     B1,0x1,A3       ; 3
 9           NOP        5               ; 4,5,6 [7,8]
10           ADD.D1     4,A10,B4        ; 9
```

When VLIW compilers parallelize the sequential source code, they introduce certain *features* that are valid only if the program is executed on the target VLIW processor but introduce data and control hazards if we sequentialize these instructions for execution on the native machine. Three types of data hazards are commonly recognized in literature that should be considered during such a *sequentialization* process.

- A *Read After Write* (**RAW**) hazards results when an instruction *uses* the results that are not yet available. Such hazards result from the existence of variable length pipelines, where different instructions may take different number of pipeline cycles to complete. Most of the instructions consume a single CPU cycle *i.e.* have zero delay slots[5], however,

[5]Delay slots, as referred in TI's literature, are the number of additional CPU cycles required for completing an instruction execution.



(a) Phases of the C6x Processor Pipeline

(b) Single Cycle Instruction Pipeline

(c) Two Cycle Instruction Pipeline

(d) Four Cycle Instruction Pipeline

(e) Branch Instruction Pipeline Phases

Figure 3.11: Pipeline Stages and Delay Slots of TI C6x Series Processors [Tex10]

quite a few instructions require *non-zero* number of delay slots. An instruction issued *before* another instruction, does not necessarily finish its execution before the later one *i.e.* out-of-order completion is possible. Figure 3.11 shows the generic pipeline stages and a few delay slot examples for C6x processors.

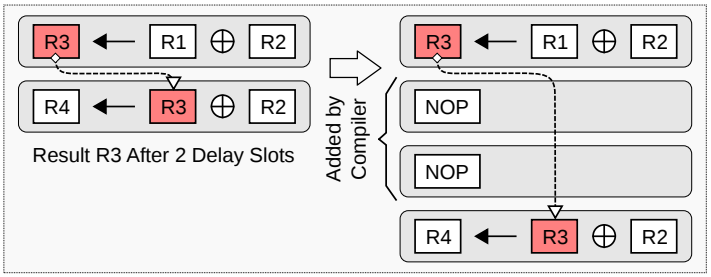


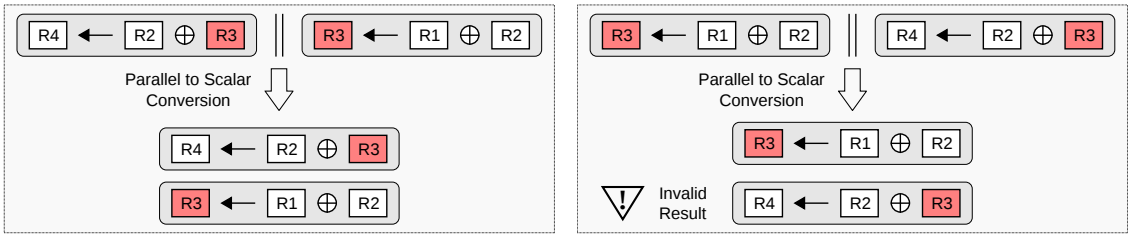
Figure 3.12: RAW Hazards in VLIW Software

At lower optimization levels, VLIW compilers handle RAW hazards by inserting *No Operation* (**NOP**) instructions, so that appropriate number of cycles pass before a given instruction tries to use the results of previous ones, as shown in Figure 3.12. As a result, such hazards require proper *handling* of NOP instructions on scalar machines such as x86 processors, usually by advancing the pipeline stages of previously issued instructions.

At higher optimization levels, VLIW compilers fill the delay slots with potentially useful but independent instructions. As an example, a branch in C6x processors takes a single execute cycle and five delay slots before executing the branch target instruction, as shown in Figure 3.11(e), if the branch is actually taken. Considering that an execute packet can contain upto eight parallel instructions on C6x processors, a maximum of

forty (40) instructions can execute in the delay slot range of a single branch instruction. Also, the C6x pipelines are never flushed *i.e.* once an instruction enters the pipeline it completes its execution.

Multi-cycle NOP instructions or multi-nops are commonly used to fill the delay slots of multi-cycle instructions. The processor effectively stops execution, except for advancing the pipeline stages of already issued instructions. Multi-nop instructions could also terminate earlier than their specified number of CPU cycles. Such situations usually arise when multi-nops are issued after branch instructions. Line 4 in Listing 3.7 is a conditional branch, which if taken will force the *Early Termination* of instruction at Line 9 after the 6th CPU cycle, otherwise the multi-nop will complete its execution and instruction at Line 10 will execute in the 9th CPU cycle.



(a) No WAR Data Hazard (b) With WAR Data Hazard

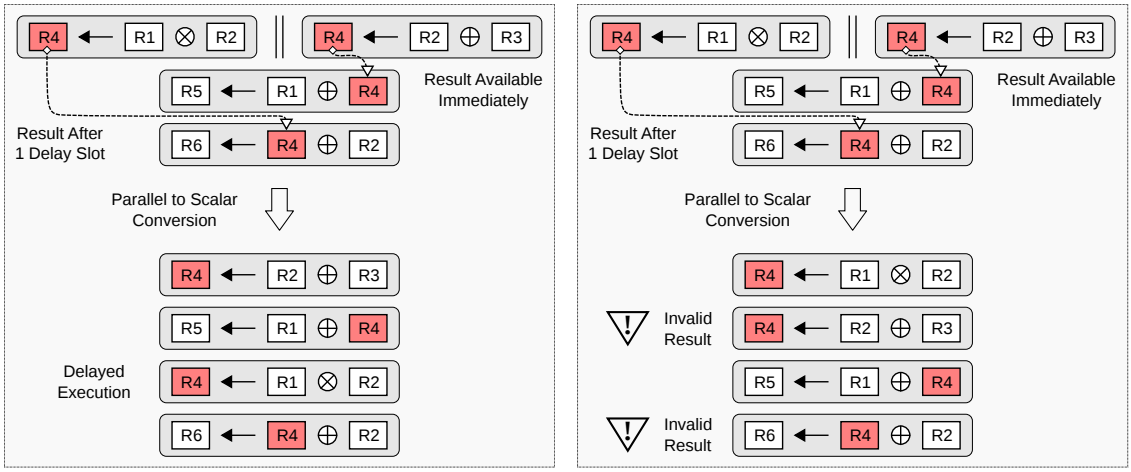
Figure 3.13: WAR Hazards in Parallel to Scalar Conversion

- A *Write After Read* (**WAR**) data hazard represents a problem with concurrent execution. In an execute packet a register can be the source and destination operand of multiple instructions simultaneously. On a parallel machine, this feature does not pose any problems irrespective of the order of instructions in an execute packet, as shown in Figure 3.13. At the beginning of an execute packet, all instructions within this packet *see* the same register state of the processor. On a scalar machine these instructions will execute *one-after-another* and in the first case, shown in Figure 3.13(a), the WAR data hazard does not appear but in the second case, shown in Figure 3.13(b), we get an invalid result. This requires that source operands of each execute packet be preserved during its execution on a scalar machine. As a concrete example, we can see a WAR hazard in Listing 3.7 between instructions at Line 1 and 3 for register `B3`.

- The *Write After Write* (**WAW**) data hazards usually result from the concurrent execution of instructions that write to the same destination operand. In such cases, the actual availability of results in the destination operand differ, as instructions have different delay slots. These hazards appear when instructions are scheduled within the delay slot range of multi-cycle instructions, as shown in Figure 3.14 where two instructions write to the same destination operand and the following execute packets use the values of operands, as they become available.

  On scalar machines the *effect* of instruction execution should be delayed, as shown in Figure 3.14(a). A simple parallel-to-scalar conversion, as shown in Figure 3.14(b), would evidently produce invalid results. As a concrete example, the `SHR` instruction at Line 8 in Listing 3.7 uses register `A3` as destination operand, which is also destination

of MPYU instruction at Line 6 but lies within its delay slot range. So the order of execution or result assignment is important on a scalar machine.



(a) No WAW Data Hazard

(b) With WAW Data Hazard

Figure 3.14: WAW Hazards Resulting from Instruction Scheduling within Delay Slots

Control hazards result from the pipeline-specific semantics of branch instructions on VLIW machines. Branch instructions do not take effect immediately but require a certain number of delay slots. Nested branch instructions are also possible *i.e.* when a branch instruction appears within the delay slot range of a previous branch instruction. As discussed earlier, once an instruction enters the processor pipeline, it finishes its execution.

Figure 3.15 shows control flow on a hypothetical VLIW machine where branch instructions take effect after three delay slots. Two branch instructions are shown, BR_1 and BR_2 in execute packets (E_0) and (E_2), respectively. The second branch instruction appears within the delay slot range of the first branch instruction. Depending upon the result of branch evaluations and once the delay slots have been consumed, control flow could take place in four possible ways, as summarized in Table 3.1. Apart from the simplest case when both branches are not taken, control flow is convoluted to some extent and becomes interesting when both branches are taken. In this case, the first branch modifies the control flow after its delay slots have been consumed (A), but the second branch is also in the pipeline, requiring its delay slots to be accounted for at the destination of first branch instruction (C), before its effect could take place (B_2). As a consequence of such control flow semantics in nested branch instructions, the control flow should finally reach the target instruction of the last taken branch instruction *i.e.* execute packet (E_m) or (E_p) in this example. On more realistic VLIW machines, such as TI's C6x series, branch instructions take effect after five delay slots and compilers can potentially schedule a branch instruction in each of these delay slots, resulting in control flows that are considerably more intricate.

3.6.2 Memory Addressing in Translated Software

The *target vs. host* addressing issue was discussed in Section 3.5.3 where our objective was to show the address-space differences in native host memory references and simulated address-space in hardware models. This issue appears in the context of static translation of VLIW

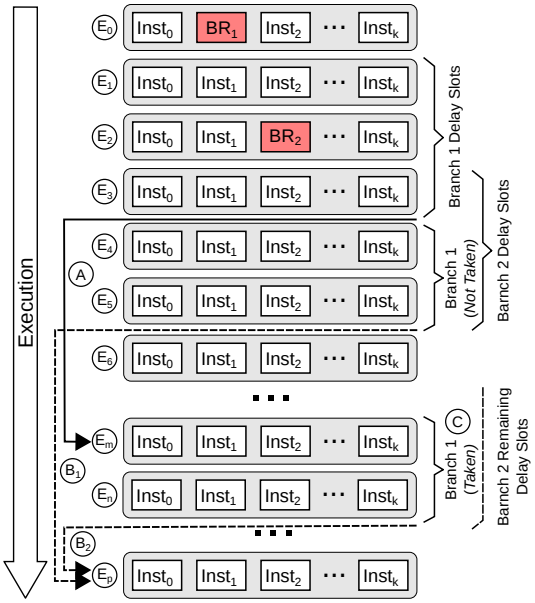


Figure 3.15: Control Hazards in VLIW Processors

	BR_1 **Not Taken**	BR_1 **Taken**
BR_2 **Not Taken**	$E_0, E_1, ..., E_6, ...$	$E_0, E_1, ..., E_3, E_m, E_n, ...$
BR_2 **Taken**	$E_0, E_1, ..., E_5, E_p, ...$	$E_0, E_1, ..., E_3, E_m, E_n, E_p, ...$

Table 3.1: Possible Control Flows for Nested Branch Instructions

binaries as well, for their native execution. We cannot transform target addresses to native counterparts, as there is no relationship between them, thus requiring a mechanism that could use them in a transparent fashion.

Data memory accesses suffer from the address-space differences only, where as instruction memory accesses incur an additional overhead of non-availability at translation time *i.e.* indirect control flow instructions. These references become known at runtime, requiring either some dynamic translation or interpretation support or static translation of all possible destination addresses.

VLIW architectures also support different addressing modes to access data memories. For example, the C6x processors support Linear and Circular addressing modes for address calculation, depending on the contents of `AMR` register. Implicit modification of source operands is also possible during address calculation, in addition to the destination operand(s) of an instruction. Line 2 in Listing 3.7 shows a side effect update for register `B5`. These types of instructions produce multiple outputs, usually with different delay slots.

3.6.3 Software Execution Timings and Synchronization

The native software executes without the knowledge of hardware models, and software annotations or similar mechanisms are required in the compiled or translated software. In

case of binary translation, precise static timing information (instruction latencies, pipeline effects, *etc*.) is easily available in native software as the final target instructions are used for translation. Nevertheless, an annotation-like mechanism is needed for synchronization with the underlying hardware platform components.

Time modeling of dynamic effects such as cache memories, requires either the use of heuristics as suggested in [GHP09], for example 90% cache hit-rate, in order to produce faster simulations. A predefined cache model could also be integrated with the translated software, if the original target addresses are preserved in the translated software, producing accurate but slower simulations. Data *vs*. instruction cache issues will be investigated. Instruction cache accesses could be different as the translated software executes instructions that are different from the original ones.

3.6.4 Hybrid and Heterogeneous MPSoC Simulation

Integration of simulation models that use different technologies such as combining dynamic translation and native techniques in a single platform could provide some interesting capabilities. Such as modeling of different processors, shared memory accesses and re-use of existing platforms for modeling complex MPSoC architectures.

A simulation platform that combines different simulation techniques would be referred to as a hybrid simulation platform. We will also investigate the integration of source compiled and binary translation based simulation techniques to model heterogeneous architectures, as shown in Figure 3.3, and will refer them as heterogeneous MPSoC simulation platforms.

3.7 Conclusion and Key Questions

Native simulation of software is a difficult problem although it might look elementary, as it involves different types of problems that do not appear in ISS based TLM models and Cycle Accurate platforms. In particular, the address-space differences in target and host machine representations and the reflection of execution timings in the simulated software *w.r.t*. to the real platform.

In case of VLIW machines, lack of supporting tools further aggravate this problem and demand a different approach that could handle VLIW specific details, as well as being efficient. Here are the key questions that we intend to answer in this thesis:

1. How can we efficiently support the simulated target address-space on host machines without requiring dynamic linking/patching support ? More specifically we want to:
   (a) Support the use of SystemC-based hardware IP models, without requiring modifications for address mappings and symbols resolution.
   (b) Minimize software coding constraints, such as the ability to use hard-coded addresses and constant link-time symbols.
   (c) To be able to simulate complex operating systems that make use of MMU based virtual to physical address translations.
2. How can we define an automatic software annotation technique into the proposed approach for MPSoC performance estimation ?

3. How can we support native execution of VLIW software, without requiring any runtime support ? More specifically we would like to emphasize on:
   (a) A VLIW simulation approach, which is generic enough and can also be applied to RISC machines.
   (b) A source-free approach, requiring optimized VLIW binaries only for generating native simulators.
   (c) Accurate support for performance estimation of VLIW software.

Research is what I'm doing when I don't know what I'm doing.

Wernher von Braun

4

State of the Art: On Software Execution for MPSoC Simulation

Native software execution provides the ability to perform early validation of embedded software, thanks to the higher simulation performance. The benefits in simulation speed are largely obtained by the *direct host* execution of embedded software, avoiding runtime software interpretation and translation overheads. A major drawback of native simulation techniques is the *almost* entire absence of target-architecture *knowledge* in host compiled software, which finds its roots in the very high level modeling of target machines and results in functional simulations. In the absence of target-specific timing information, architecture exploration is not only difficult but rather *impossible*, undermining the utility of native techniques.

In order to introduce target-specific temporal information in native software, many annotation techniques have been proposed in literature. Such techniques are usually based on retargetable compilation frameworks that could generate software binaries for both target and host machines. Specifically, in case of *Very Long Instruction Word* (VLIW) [Fis83, Fis09] processors, such frameworks are rare and even if available, are not mature enough to match the vendor-supplied toolchains, in terms of optimization. In such situations, using annotation techniques will almost always produce very inaccurate results and would be rarely useful for *Design Space Exploration* (DSE) purposes. Our focus will be to preserve target-architecture *knowledge* as much as possible, either using compilation or static binary translation, so that the resulting simulators reflect precise target processor timings.

4.1 Native Platforms for MPSoC Simulation

Native simulation of software is a well-known concept, ranging from primitive techniques based on the host execution of algorithms for functional verification to more advanced implementations relying *Hardware Abstraction Layer* (HAL) and dynamic linking concepts.

The primitive native platforms are based on the encapsulation of software tasks inside

hardware platform models. More advanced platforms profit from the layered organization and support validation of software except the HAL layer, as this layer is provided by the underlying hardware platform models. The HAL is usually replaced with a target-specific one, when the software is finally compiled for the *real* platform. Hybrid and address remapping/unification techniques have also been proposed in literature. We start our discussion on native platforms by reviewing software encapsulation based techniques.

4.1.1 Software Encapsulation

In its most general formulation, native software simulation targets the direct execution of software code on the host machine with the use of a wrapper to connect to an event-driven simulation environment. Initial proposals suggest to encapsulate the application code into TLM modules using hardware threads, as if they were implemented as hardware IPs [GCDM92, GYNJ01, BYJ02, WSH08, CHB09a]. Only a subset of software tasks are encapsulated within the hardware modules *i.e.* processing elements of the platform, as shown in Figure 4.1. The hardware simulation kernel schedules both hardware and software threads, bringing implicit and unintended concurrency to the software tasks and no way to account for the operating system behavior.

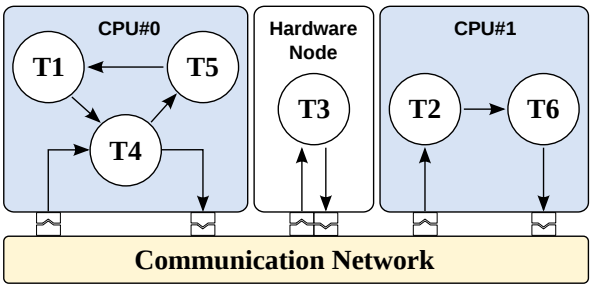
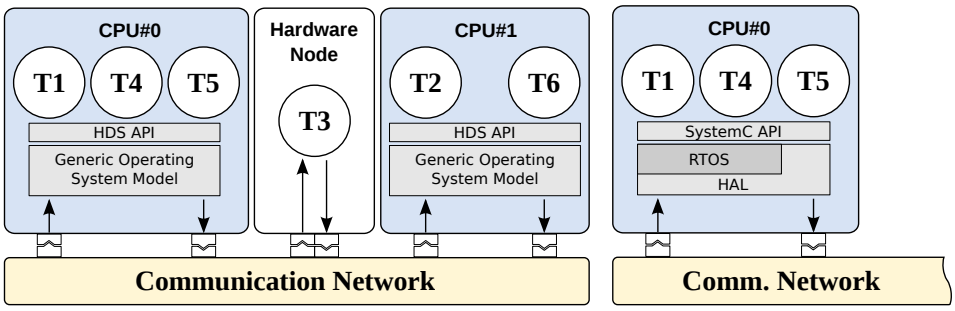


Figure 4.1: Software Encapsulation in Native Simulation

These solutions are simple but suffer from two severe drawbacks. Firstly, the simulated code provides a very limited form of parallelism *i.e.* co-routines. Secondly, since it executes in a hardware module, all data allocations by the software are indeed allocated within the simulator process address space instead of the simulated target platform memory. There is no way for a platform device to access a buffer allocated by the software because the buffer is not visible to it, and should it be, the addresses of simulated platform have no relationship with the addresses of the simulator process. Moreover, the software execution is constrained within the hardware context, such as how it accesses the underlying platform resources using the platform interfaces *e.g.* ports in TLM models. These approaches are clearly not suitable for supporting legacy code.

Due to the complexity of embedded software and execution dependency requirements, integration of abstract OS models directly into the native simulation environments have been proposed. The goal is to provide an implementation of a lightweight operating system using the event based primitives of the simulation environment, thus each software task becomes a hardware module. Using this approach, the modeled RTOS relies on the scheduler of the hardware simulator instead of the scheduler that the RTOS would use, even though some solutions suggest to modify the hardware simulation kernel for this purpose. These works realize some RTOS scheduling algorithms inside the simulated hardware threads to execute

application tasks, with different constraints and priorities. A set of HDS APIs including task creation, *Inter Process Communication* (**IPC**) and semaphore services are implemented to adapt the requirements of application tasks [GCKR12], as shown in Figure 4.2(a). A few of such solutions are based on SpecC [GYG03, SD08], whereas most others use SystemC [YNGJ02, MVG03, LMPC04, PAS+06, NST06, EHV09, HSAG10] as the platform modeling language. Some have even tried to model multiple Operating Systems (GPOS and RTOS) using this technique [PVRM10]. Unfortunately, these OS models are not detailed enough as all of the C library calls (among others), including memory management routines, are outside the control of OS model. This limitation makes the device driver development impossible using such models. Similarly, as these models require rewriting of application software, they effectively prevent use of legacy code. Some instances of proprietary co-simulators [HWTT04] have also been proposed that make use of the encapsulation concept.



(a) With an Abstract Operating System Model (b) With an Actual RTOS

Figure 4.2: Software Encapsulation with an Abstract/Real Operating System

To improve the realism of software execution, some works have proposed to use an actual RTOS instead of an abstract OS model, as shown in Figure 4.2(b). For example, the SPACE platform [CBR+04] uses an actual RTOS *i.e.* μC/OS-II and provides a SystemC-based API layer to software tasks instead of the traditional HDS API. The software stack is still encapsulated within the hardware models, but the OS is the same as it would be on the *real* platform, resulting in certain advantages *e.g.* the software threads are scheduled by the real RTOS, providing realistic modeling of the real-time software properties. Similarly, such frameworks provide easy means for architecture exploration as they support different abstraction levels and allow modification of component types from hardware to software and vice versa. Lastly, such models enable the execution and validation of a significant amount of software using native simulation.

In any variant of software encapsulation, the simulated software is strongly bound to the platform models, thus modeling of SMP-like task migration is impossible. Additionally, the amount of software that could be validated is limited and needs to be modified to *accommodate* the low level platform interface requirements. In such techniques, the native software executes independently on the host machine and the hardware platform has no control over it, until the software explicitly returns control to it by invoking low level platform service(s). In between the low level interaction points, native software executes *atomically w.r.t.* the hardware platform and accesses the host machine resources directly. This problem has been addressed in [KKW+06, WH09, CHB09b, GHP09], which propose different annotation techniques to insert function calls in software, for modeling such accesses to

platform resources.

4.1.2 Hardware-Software Interfaces

Abstraction is defined as a simplification process where only the essential details of a complex entity are kept, for a specific objective. The abstraction concept can be applied to software, as well as hardware components of a given MPSoC system. *Cycle Accurate* (**CA**), *Transaction Accurate* (**TA**), *Virtual Architecture* (**VA**) and *System Level* (**SL**) are the most commonly recognized abstraction levels for hardware modeling. Similarly, the software could be simulated at *Instruction Set Architecture* (**ISA**), *Hardware Abstraction Layer* (**HAL**), *Hardware Dependent Software* (**HDS**) or Functional abstraction levels.

A hardware/software interface serves as a virtual machine where it executes commands *e.g.* instructions, functions *etc.*, from the software and interacts correctly with the hardware platform. Many hardware/software interfaces are possible in MPSoC simulation environments [BYJ04, SGP08, GGP08]. We focus on *Transaction Level Modeling* (**TLM**) [CG03] of hardware components and discuss the most commonly used software abstraction levels in TA platforms and their interfaces *i.e.* HDS ❶, HAL ❷ and ISA ❸, as presented in Figure 4.3.

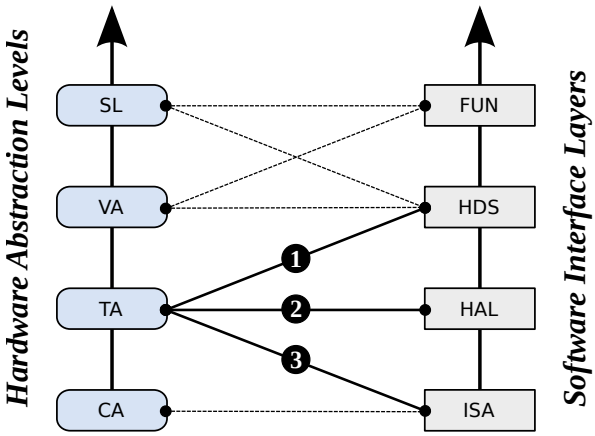
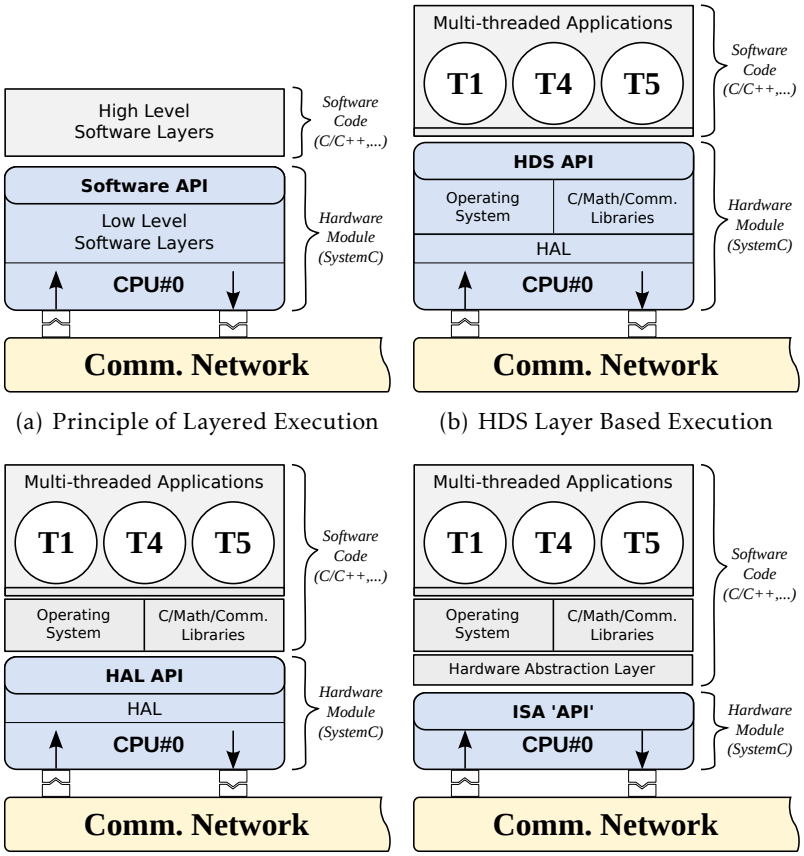


Figure 4.3: Hardware Abstraction Levels and Software Interface Layers

Layered software execution profits from the fact that hardware platform can provide a real API for interacting with the software world. This API defines the interface level and is usually implemented inside the hardware models using a software programming language. This interface API makes the higher software layers completely independent from the lower hardware models, as they can be compiled and executed on top of the provided interface. Figure 4.4(a) shows the principle of such execution models.

Most frequently used layered software models [BBY+05, TRKA07, PJ07, CSC+09] implement the HDS API for higher level software applications. It is a difficult solution because to build realistic applications using such high level interface, the hardware models have to implement *numerous* APIs. Theoretically speaking, this abstraction level has to provide all APIs for the operating system (modeled or real) and the standard software libraries, such as C and Math libraries. However, in reality such interfaces implement only a small subset of these APIs, imposing constraints on the application software. The amount of software that could be validated using such interfaces is very limited, thus undermining their practical use. In certain cases, the standard C API functions available on the host machine are directly used,



(a) Principle of Layered Execution (b) HDS Layer Based Execution

(c) HAL Layer Based Execution (d) ISA Layer Based Execution

Figure 4.4: Layered Software Execution at Different Interface Levels

thus the simulated software executes outside the framework of hardware platform resulting in inaccuracies. Moreover, none of these approaches target dynamic task creation or thread migration between processors, as it takes place on SMP platforms. Figure 4.4(b) shows the organization of HDS interface-based simulation platforms.

A few approaches [YJ03, YBB+03, BYJ04, GGP08] rely on the definition of a thin HAL layer that must be used for all hardware related accesses. The HAL layer is implemented in a wrapper that includes a hardware thread per processor, commonly known as *Execution Unit* (**EU**) (Figure 3.6). Entire software stack above the HAL can be compiled independently, including the OS and standard libraries, and executed natively. Each HAL function call is performed in the context of an EU assuming that all EUs belonging to the wrapper share the Operating System code and data structures. As the context switching function belongs to the HAL layer, SMP-like software thread migration is supported in such platforms. Figure 4.4(c) shows the structure of HAL interface-based designs.

Native simulation platforms that are based on the *so-called* ISA 'API' (or Host ISA Layer), as shown in Figure 4.4(d), have not been discussed in literature. Most common TA platforms which provide an ISA level software execution use target *Instruction Set Simulators* (**ISSes**) instead, as discussed in the previous chapter (Section 3.4.4.1). The principle deficiency of such models emerges from the use of either ISSes [LP02, HJ08] or *Dynamic Binary*

Translation (**DBT**) based solutions [GFP09], which interpret/translate instructions at run-time and result in slower simulations.

All of the native simulation techniques discussed so far suffer from address spaces problem where the native software executes inside the *host* address space and the hardware platform simulates the *target* address space. These differences make it difficult to model certain hardware/software interactions, as explained in the previous chapter (Section 3.5.3). Two main classes of solutions have been proposed to solve this issue *i.e.* address remapping and address space unification.

4.1.2.1 Address Remapping Techniques

Simple remapping techniques perform address conversion between the target address space and the simulation process address space for I/O accesses identified by the use of specific primitives. This does not solve the issue of external accesses to natively allocated buffers. More complex remapping strategies rely on the fact that a *host* OS exception will be raised [PV09] when an access to a bad virtual address takes place. The principle is to mark as *invalid* all memory pages visible to the platform components, using the *host* operating system. Any access to these pages will raise an exception that the simulator process can *trap* and *handle*. There may be performance issues of this technique, if many exceptions are raised, and the technical aspects of handling *overlaps* between both memory spaces remain a problem. For example, the memory addresses over 3GB (`0xC0000000`) in the x86 architecture are reserved for Linux kernel code only.

Yet another work [PDV11] uses *host*-to-*target* address conversion for data cache modeling. This approach requires the sizes of native variables to be the *same* as in the target platform, and accomplishes this task by implementing an XML-based code builder extension. Moreover, it lacks support for dynamic memory allocation and would require the creation of a heap manager for this purpose.

4.1.2.2 Uniform Memory Representation and Dynamic Linking

Unification relies on the use of a unique memory mapping for the software, as well as hardware components of the native simulation environment [Ger09]. The simulator process mapping is selected for this purpose as it is also the one used by the simulated software stack. Unification requires to modify the hardware platform so that each IP exports a set of *mandatory* symbols to be resolved at linking time to perform a low cost remapping. Such platforms also require modifications the OS, so that it accesses the hardware solely through HAL function calls and never uses hard-coded addresses.

The drawbacks include modification of the simulation models to build the unified memory space, the addition of a specific linking stage visible to the user, and the OS port on the native HAL layer, as discussed in Section 3.5.4. In order to simulate the full C library, especially the memory management functions, a well separated address space should be provided to the natively executing OS kernel.

4.1.3 Hybrid Techniques

Some researchers have proposed hybrid solutions that are partially based on native execution. These solutions try to mix two different technologies together to improve on accuracy such as offered by an ISS, while profiting from the higher simulation performance using native

execution. We can refer to [BPN+04, MRRJ05, KGW+07, KEBR08] as key propositions in hybrid simulation platforms.

The HySim framework, initially proposed in [KGW+07], is composed of two key components *i.e.* an ISS and a native processor. An external control logic manages the switching between ISS and native processing elements but allows this switching to take place at function call boundaries only *i.e.* a function simulated on the ISS may call a native function or vice versa. *Target-independent* source code is compiled for the *host* machine using native compiler and instrumented for defining the synchronization points, as well as performance estimation. Rest of the software stack is compiled for the target machine and executed on the ISS. Figure 4.5 shows the latest version of HySim framework as it appears in [MEJ+12], where Target Simulator (TS) and Abstract Simulator (AS) are the ISS and native processing elements, respectively.

A key problem in hybrid approaches is the control mechanism, which selects the execution mode *i.e.* whether use the ISS or native processor for a given function during simulation. In [KGW+07, GKL+07] stubs are used to translate function calls from ISS to native processor and vice versa. These stubs are directly inserted in the instrumented code that runs on the native processor, whereas the code running on ISS is monitored and inverse stubs are invoked. The monitoring process is accomplished by the use of a *control logic*, also known as *Control Layer*, as shown in Figure 4.5.

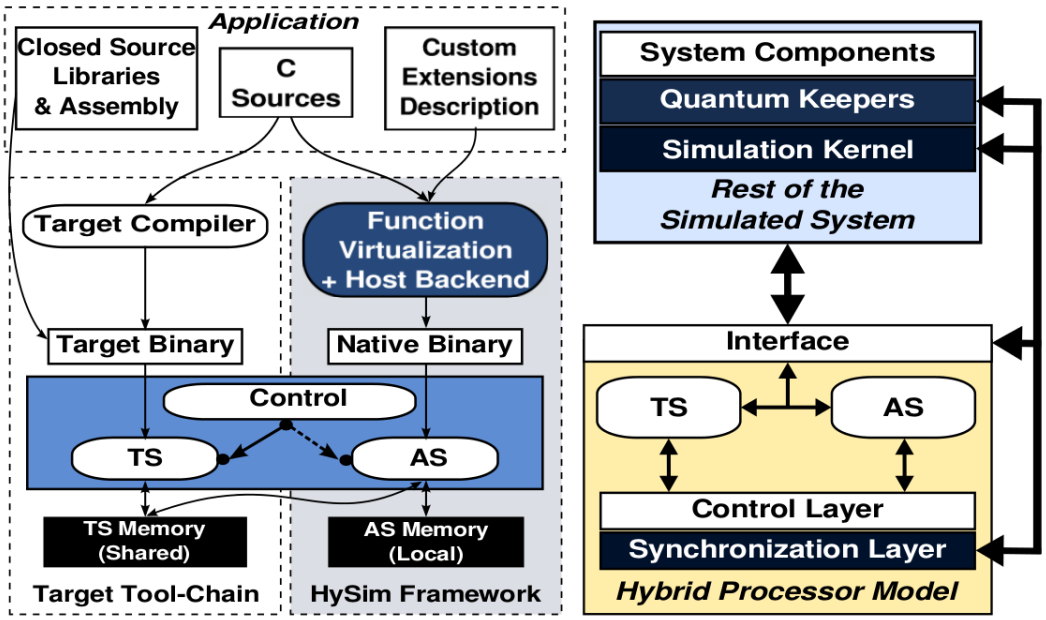


Figure 4.5: Architecture of the HySim Framework [MEJ+12]

The HySim platform as in [KGW+07, GKL+07, GKK+08] and other hybrid frameworks suffer from *hybridization-introduced decoupling*, as native processors and ISSes could become decoupled as a result of simulation speed differences. This limits the use of such MPSoC simulation systems where reactive behavior and interprocessor interactions do not define the system correctness and usability. Yet another type of decoupling referred as *temporal decoupling* allows some of the components in a simulation platform to run ahead from rest of the system, as they do not frequently interact with their environment. Temporal decoupling

is used to avoid unnecessary kernel synchronization points and context switches, which usually cause significant overhead. In order to guarantee the correctness of a decoupled MPSoC simulation environment, *hybridization-introduced* and *temporal decoupling* issues have to be addressed as proposed in [MEJ+12], where they introduce a Hybrid Processor Model including a Synchronization Layer to manage these issues, as shown in Figure 4.5.

In [MRRJ07a] calls to the native functions are statically inserted into the code interpreted by the ISS, using a mechanism similar to *Remote Procedure Call* (**RPC**). The native function selection is based on certain performance and energy estimation constraints. They try to minimize the total simulation time under a maximum constraint on the error in energy estimation, as proposed in [MRRJ07b].

Key limitations of the hybrid simulation platforms lie in the complexity of managing interactions between fast native processing elements and the slow ISSes. Nevertheless, such models have good precision as architecture-specific details are taken into account using the low level simulators. However, the simulation performance remains a bottleneck providing around an order of improvement for MPSoC simulations as compared to the ISS only simulations.

4.2 Performance Estimation in Native Simulation

Many of the recent simulation approaches [KKW+06, TRKA07, HAG08] are based on the native execution of software to take benefit from considerable simulation speedup. However, most of them partially handle different sources of timing dependencies, to allow for reliable performance estimation in a multiprocessor context.

The basic idea behind performance estimation in native simulation is to introduce target specific performance metrics in the host compiled software. Even though the software is executed directly on the host machine, it still *reflects* the timing behavior of target processor architecture. These performance metrics, commonly inserted in software using *annotations*, can be introduced by analyzing software at different abstraction levels. Software performance estimation techniques are classified as:

- Source Level Simulation (SLS) (also known as System Level Simulation)
- Intermediate Representation Level Simulation (IRLS)
- Binary Level Simulation (BLS)

Many automated approaches like [LP02, CGG04] as well as manual ones [PMP+04] rely on the insertion of timing information in the software. This timing information usually takes into account memory accesses as addressed in [KKW+06] and sometimes cache memory effects as in [CGG04], but it seems that none of them considers memory accesses generated by the software execution itself. Specifically, access to the program memory are ignored, while instruction cache size may be limited due to integration constraints and this may lead to non-negligible conflicts in multiprocessor context.

In the following sections, we review the existing performance estimation techniques within the context of RISC and VLIW machines.

4.2.1 Source Level Simulation

Source level estimation techniques [GMH01, KKW+06] introduce annotations to the original software sources, in order to get approximate performance estimates. In most cases, these

estimations encompass only a certain portion of source code for estimating it's execution time, generated memory accesses or even energy consumption on a particular target machine.

Source-based performance estimation technique introduced in [BHK+00] converts machine instructions into an abstract "C-view" of the assembly code using partial compilation for target machine or a virtual instruction set using a virtual compiler. The number of processor cycle estimates are specified using a processor description file instead of directly inserting them into the annotations. Performing estimations for a different target processor requires only to change the processor description file(s). These approaches are similar to native software execution as the target cross-compiled code is finally converted into host machine code. However, the executed software does not contain debug information for validation and has a structure that is far from the original one. Similar techniques have been proposed in [GMH01] and [MSVSL08].

The techniques proposed in [SBVR08, WSH08] compile the source code for the target processor and then use debug information to establish correspondence between the generated binary code and the original sources. These solutions remain complex as the structure of generated code is very different from the original sources when the compiler optimizes the code significantly.

The technique proposed in [PHS+04], does not directly instrument the software but instead relies on *polymorphism* and *operator overloading* features of C++ language. They overload all basic operators in C++, resulting in automatic insertion of performance annotations during compilation and native execution of software. This solution seems reasonable in terms of performance estimates but cannot cope with target processor specific optimizations.

Such native approaches can provide application and OS timing information by embedding simulator `wait` statements into the software code. However, most compiler optimizations lead to executable code whose execution paths are not isomorphic to the ones of the source code. This aspect is further aggravated in VLIW binaries, as the resultant code is parallel in nature; thus, finding a match between the original sources and final binaries is much more difficult. Source level annotations are therefore intrinsically inaccurate, giving at best rough runtime estimates.

4.2.2 Intermediate Representation Level Simulation

More innovative approaches rely on the intermediate representation used by retargetable compilers. Such representations, usually some kind of *bytecode*, contain all semantic information and are used throughout the compilation process, including optimization passes.

Estimation techniques proposed in [KAFK+05, KKW+06] are based on the generation of an IR in three address code format. This intermediate representation uses the C syntax and could be directly instrumented and compiled for the host machine. This solution allows to account for the compiler optimizations on IR and the dynamic aspects of software execution.

A basic block level annotation scheme is used in [CHB09b]. The result of this annotation technique is a SystemC module containing generated C instructions of the software program along with timing estimations. The generated SystemC modules can be used in simulation platforms, in combination with the generic RTOS models introduced in [LMPC04, NST06, EHV09]. This solution takes into account the compiler optimizations, as well as target processor specific features such as branch prediction penalties and data dependencies between instructions. Although data memory accesses are considered during

instrumentation but program memory accesses are ignored that are generated during the program execution on the target machine. Moreover, this approach does not allow for debugging of the original program sources, instead the instrumented code is visible during the debug stage.

The IR based *iSciSim* approach proposed in [WH09] is an evolution of source level instrumentation approach proposed in [WSH08]. The overall design flow of the *iSciSim* approach is shown in Figure 4.6. The *iSciSim* technique is based on *GIMPLE* representation of GCC compiler, which uses a C language style syntax. The instrumentation technique relies on multiple compilation stages, where the source code is initially cross-compiled for the target processor. At the end of first cross-compilation stage, the compiler produces IR files in *GIMPLE* format that are converted to real C source code known as *Intermediate Source Code* (ISC).

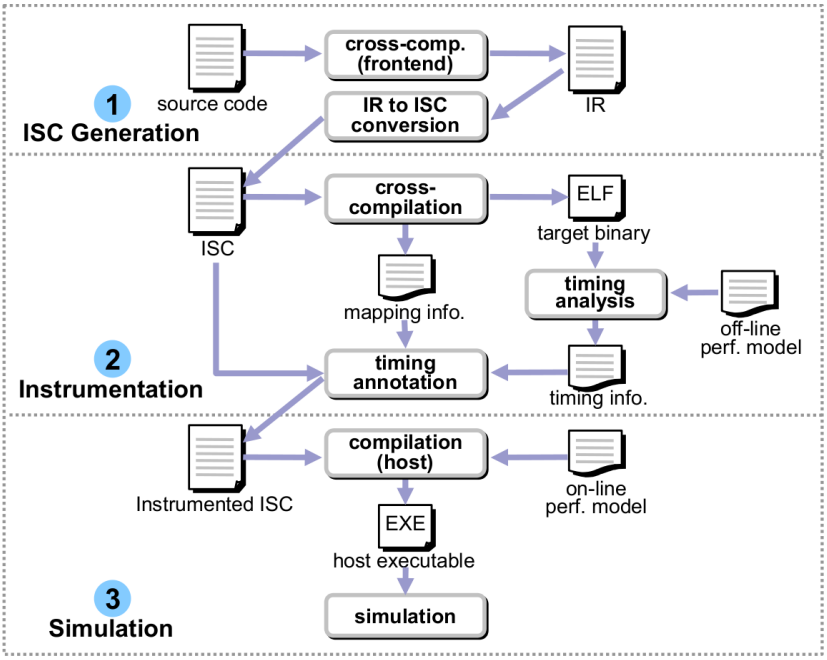


Figure 4.6: Design Flow of the iSciSim Approach [WH09]

A second compilation stage takes the ISC format sources and cross-compiles them for the target processor again. In this stage the compiler instruments the IR with target-specific metrics and generates instrumented ISC sources. Timing information is obtained from the static analysis of generated target binary and analysis of debug information allows to extract the memory mapping information, which is useful in modeling dynamic software behaviors during simulation.

Finally, a third compilation stage takes the instrumented ISC representation and compiles it for the host machine. This technique offers many advantages, including the static and dynamic aspects of software execution, such as instruction latencies, pipeline effects, branch prediction and cache memory effects. On the downside, this technique suffers from relatively low simulation speed when the instrumentation accounts for cache memory effects, as a direct consequence of high number of synchronizations with the hardware models. Moreover, the software has to be debugged using the instrumented ISC representation, as it is quite difficult to establish a correspondence between the original sources and ISC code. This

particular limitation makes it difficult to use this solution for software validation, within the context of system simulation.

A similar but more efficient technique is proposed in [GHP09], which is based on LLVM compilation infrastructure. The compiler at first compiles the source code as if the binary code will be generated for the target machine including target-specific optimizations. The compiler maintains two levels of intermediate representations *i.e. machine-independent* and *machine-specific* ones. The target backend applies target-specific optimizations on the machine-specific IR while the compiler maintains its correspondence to the machine-independent IR, known as *Cross-IR*. At the end of target compilation stage, instead of emitting code for the target machine, an annotation pass is introduced, which instruments the *Cross-IR* to produce *Annotated Cross-IR*. The annotation pass analyzes the target code and inserts calls to an *annotation* function, at the beginning of each basic block in *Cross-IR*. The second compilation phase, invokes the host backend and generates native binary objects from the *Annotated Cross-IR*.

This technique allows software debugging from original sources, as links between IR and original sources are maintained by the compiler during all compilation stages, which is an essential requirement for software validation. Moreover, this technique allows for estimating static, as well as dynamic aspects of software execution including instruction latencies, branch prediction and data dependencies. However, modeling of cache effects is difficult, as target specific memory addresses are not available in host compiled software. Additionally, architectural differences between host and target machines are difficult to account for and authors suggest to use *heuristics* for this purpose.

In order to use IR based techniques for VLIW performance estimation, the compilers need to support both host machine and target VLIW architectures. Once such a framework is available, an appropriate annotation pass could be added to annotate the target-independent IR and a subsequent host compilation would produce native software binaries, reflecting VLIW execution timings. Unfortunately retargetable compilation frameworks, such as LLVM still do not support VLIW backends, and we look into binary level simulation techniques for VLIW machines.

4.2.3 Binary Level Simulation

Binary level simulation techniques [ZM96, LBH+00] are based on the analysis and conversion of final target software binaries to native code. As these techniques start from the final software binaries, they benefit from all target compiler optimizations and produce accurate simulations. In most of these techniques, the binary code is converted to an equivalent C language representation, which is then compiled for the host machine with additional annotations to model its performance.

Simulation techniques for VLIW machines have primarily focused on the architectural and micro-architectural modeling aspects [MME+97, BBCR02, BK07, WYW08]. Most of these environments model the low-level processor pipeline details and are interpretive in nature, resulting in very slow simulations. Some of these have even been developed for specific objectives, like DLX [BK07] simulator for the Educational Purposes and offers very simplistic view of a *virtual* VLIW machine.

The VLIW hardware is relatively simple as compared to superscalar processors (Section 3.6), due to the explicit *Instruction Level Parallelism* (**ILP**) specification. Following sections briefly review the compiled simulation and binary translation techniques, with a primary

focus on how such technologies could handle simulation and performance estimation of VLIW software on native simulation platforms.

4.2.3.1 Compiled Simulation and Static Translation Techniques

The key idea that differentiates compiled simulation from interpreted ones, is the decoupling between instruction fetch/decode steps and its execution. Figure 4.7 shows the basic principle of such approaches, where a translation front-end decodes the target binary instructions and generates an IR, which is then compiled for the host machine using native compiler. The compiled simulator is loaded into the host machine memory and an execution runtime, calls the instructions behaviors during simulation.

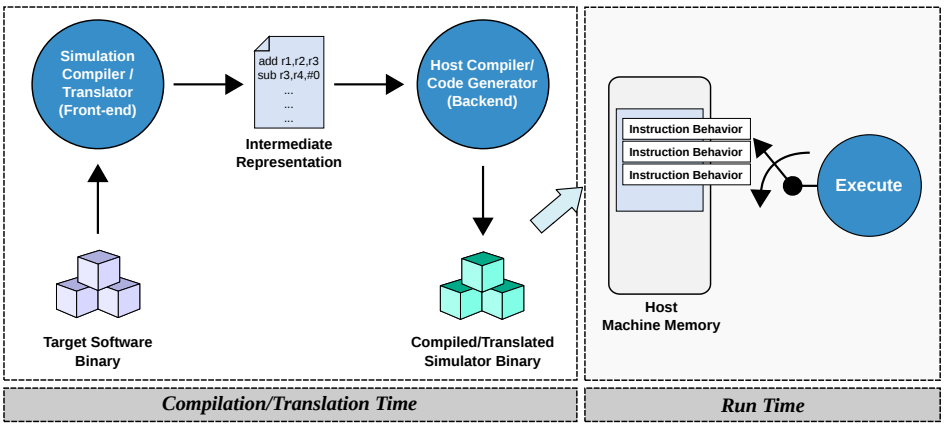


Figure 4.7: Basic Compiled Simulation/Static Translation Principle [NBS+02]

The very first compiled simulation approach was proposed in [MAF91], which describes how a high-level language could be used as IR for modeling ISA of RISC machines. Initial compiled hardware/software co-simulation approach for programmable DSP architectures was proposed in [ZTM95]. Rather than using a high-level language as the code generation interface, [ZG99] improves the compiled simulation by using the host machines resources and introducing a low-level code generation interface specialized for ISA simulation.

Many similar techniques have appeared in literature, such as [LEL99, LBH+00, RMD03, BG04, NTN06]. Generation of both compiled and interpreted simulators has been proposed in [LEL99], using a macro-based library of simulation functions. Key limitations include, simulation of single instruction per cycle and construction of a topological sort order for handling parallel instruction dependencies. Auxiliary variables are introduced to break such cyclic dependencies, by storing the register contents temporarily. The *Instruction Set Compiled Simulation* (**IS-CS**) technique proposed in [RMD03] uses a template customization approach to instruction set simulation. Target binary instructions are decoded one-by-one and custom templates are generated. These templates are subsequently compiled for the host machine. Runtime behavior is very similar to an ISS, where each instruction is examined before execution and redecoded, if necessary.

Many of these approaches fail to handle dynamic code behaviors, such as indirect branch instructions and provide a *fallback* mechanism using interpretive mode simulations or use techniques that slowdown the simulation speed. For example, SuperSim [ZM96] and [LBH+00] treat every instruction as a possible target of an indirect branch instruction and

thus set a label before the translated code of each instruction. These labels limit the number of optimizations that could be applied during host code generation, as the compiler must assume that each label could be an entry point for control flow. Similarly, SyntSim [BG04] and WSS [NTN06] propose to give control to an ISS, when an indirect jump is found during simulation.

Compiled simulation benefits from much higher simulation performance, as the runtime overheads of interpretation are avoided but suffer from the inability to handle dynamic code behaviors, and frequently includes compiler-unfriendly features, as discussed in [NTN06]. For example, most of the techniques generate a single simulation function containing a very big switch structure with many labels. Such code adversely effects the compilation performance and limits the number of optimizations that could be applied during code generation. Moreover, features like *breakpoints* and *fast-forwarding* are difficult to implement in such systems. None of the solution is efficient, within a hardware/software co-simulation environment, as *target* to *host* address mappings require a memory model and frequent accesses to such a model significantly reduce the simulation speed.

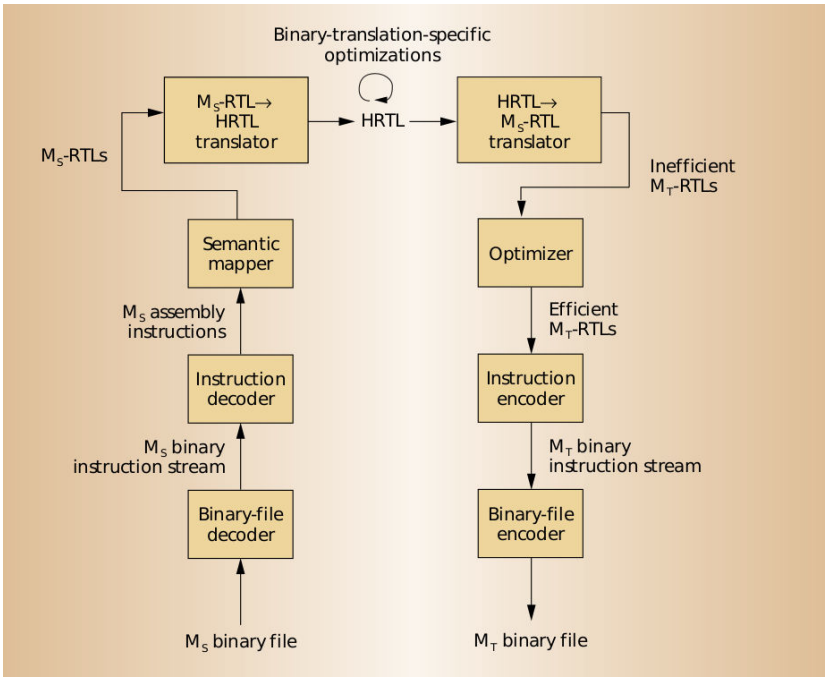


Figure 4.8: Static Binary Translation Flow in UQBT [CVE00]

Static binary translation techniques like FX!32 [CHH+98], UQBT [CVE00], [CYH+08] and LLBT [SCHY12] primarily focus on scalar architectures and porting of a given ISA to another platform. FX!32 translates Windows NT/Intel instructions to run on the Windows NT/Alpha platforms. Initially, an interpreter gathers some execution profiles that are subsequently used to guide the static binary translator for generating native Alpha code. Later executions of the same application can re-use the already generated native code. Another static binary translation approach known as UQBT [CVE00], focuses on providing adaptable binary translation. UQBT introduces HRTL, an intermediate representation, which raises the abstraction level of input binary instructions and makes them source[1] independent.

[1] In binary translation domain, *source* refers to the input machine architecture and *target* usually means the

Thus, allowing for the subsequent generation of native code that uses the native *Application Binary Interface* (**ABI**) for improved performance. UQBT integrates an interpretor as well, to handle the dynamic aspects of translated code. As the translation framework is static, support for kernel code and dynamically linked libraries is not available. Figure 4.8 shows the overall binary translation flow in UQBT.

A direct ISA-to-ISA binary translation system is proposed in [CYH+08] for ARM to MIPS-like machines and includes some ISA optimizations to generate efficient code. Although such direct mapping solutions could provide some specific optimization opportunities but are generally very restricted. For example, the target architecture, MIPS in this case, has to provide higher set of resources such as CPU registers, otherwise establishing a direct mapping between source and target architectures is difficult. This is particularly valid in cases where source machines, like VLIWs contain much higher number of registers as compared to the target architectures, such as x86 machines. The direct ISA-to-ISA solution in not retargetable and does not suit simulation systems, as multiple architectures need to be evaluated during architecture exploration. Additionally, by making use of an IR, we can profit from *already* available machine independent and target machine specific optimizations. For example, LLVM provides many optimizations that are very time consuming, difficult to develop and test for a direct ISA-to-ISA binary translation system.

Recently, a static translation scheme for ARM processors has been proposed in LLBT [SCHY12], which is based on LLVM IR and profits from existing optimization and retargetability features of LLVM infrastructure. This translation scheme provides some interesting ideas including jump table recovery, reduced address mapping table, PC-relative data inlining and avoiding the runtime translation requirement. The whole input binary is treated as a single function and a very big `main()` function is generated, where each instruction is labeled in the translated code, limiting optimizations during native compilation. Their focus remains on scalar-to-scalar translation, as they target IA32, Intel x64 and MIPS architectures for porting ARM applications.

4.2.3.2 Hybrid Compiled and Dynamic Simulation Techniques

Static translation and compiled simulation techniques have certain limitations, mostly relating to dynamic software behaviors, making them less interesting for general purpose computing systems. Thus, researchers are inclined towards dynamic and compiled simulation techniques such as Shade [CK94], Embra [WR96], FastSim [SL98], Aries [ZT00], JIT-CCS [NBS+02] and HIS-CS [RMD09]. All of these techniques try to combine the advantages of compiled and interpretive techniques. We briefly discuss them, as to give the reader a taste of existing techniques using dynamic principles, but our main focus will remain on static and compiled simulation techniques.

The generic principle of dynamic compiled simulation is shown in Figure 4.9, where techniques like Shade, Embra and Aries try to eliminate the compilation overhead by performing more complex instruction decoding at run-time. The key feature that distinguishes these techniques from interpreted ones, is the use of a translation cache as to benefit from the repeated execution of instructions and amortize the instruction decoding overhead. FastSim specifically refers to this feature as *Memoization*, where micro-architectural states and corresponding simulation actions are cached, for later reuse at a much reduced cost. Another benefit of such approaches, comes from the decoding of only those input

host or final execution platform.

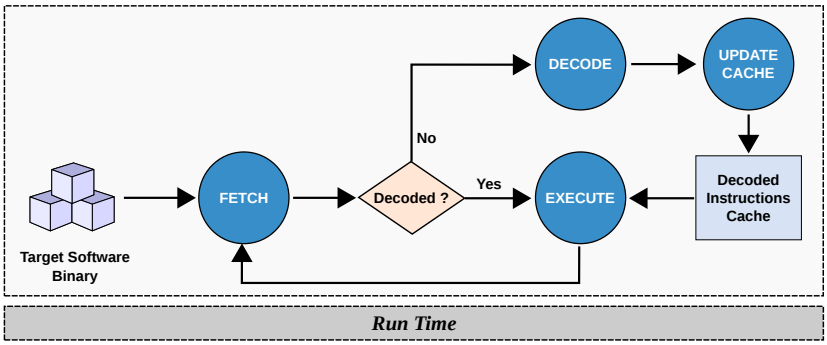


Figure 4.9: Generic Principle of Dynamic Compiled Simulation

instructions that are actually executed, as opposed to static translation schemes where the complete input binary is decoded.

Just-In-Time Cache-Compiled Simulation (**JIT-CCS**) [NBS+02, BNH+04] integrates a simulation compiler into the interpretor and compilation for the target instructions take place at runtime and results are cached. This technique improves on the ISS performance by removing the instruction decoding overhead and triggers recompilation if input binary instructions are modified. The translations are simple and unoptimized as to minimize the runtime overhead, and applicability to VLIW machines has not been demonstrated.

Hybrid Instruction-Set-Compiled Simulation (**HIS-CS**) [RMD09] takes a slightly different approach and instead of generating code for every instruction in the input binary, it generates optimized code for decoding these instructions at runtime, using a template-customization technique previously proposed in IS-CS [RMD03]. The generated decoder is specific to the input binary, as it contains decoding templates for instructions found in the given input binary only. The customized decoder along with the simulation engine is compiled using native compiler to generate custom *Hybrid* IS-CS simulator, as shown in Figure 4.10. A key feature of this technique is the reduced compilation time as the size of generated decoder is much smaller than the source of decoded program. Additionally, the generated decoder is optimized statically, as compared to fully dynamic techniques. Simulation still takes place at instruction level and VLIW simulation specifics remain un-addressed.

Binary translation techniques have retained focus on scalar-to-parallel translations, such as in the so-called *Code Morphing Software* (CMS), which targets the Transmeta Crusoe [DGB+03] microprocessor architecture for full system level implementation of x86 architecture. The Crusoe processor design consists of a hardware VLIW core and is supported by the Code Morphing Software layer, which translates the x86 instructions to native Crusoe VLIW instructions. Additionally, the translation system combines a dynamic binary translator, an interpretor, optimizer and runtime system. Similar solutions have appeared in DAISY [EA97] and Aries [ZT00]. These solutions are more complex than traditional translation systems, as they have to extract ILP from the scalar input instruction stream, so as to profit from the target architecture resources and achieve better performance.

A few efforts have focused on the simulation of SIMD instructions sets such as FX!32 [CHH+98] for translating MMX SIMD instruction sets to Windows NT/Alpha platforms. Similarly, more recent efforts from [MFP11] and [MFP12] have focused on adapting the popular QEMU [Bel05] binary translation system to support SIMD and VLIW architecture simulation on x86 hosts, respectively. These solutions are quite difficult to implement and

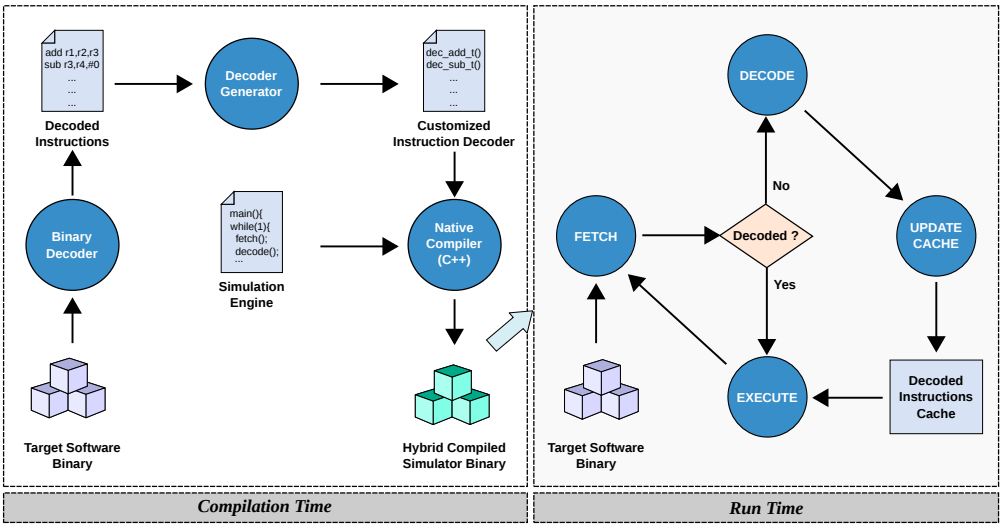


Figure 4.10: Principle of Hybrid Compiled Simulation [RMD09]

simulation performance is bound to be limited, due to the runtime translation complexity.

4.3 Discussion and Conclusions

We presented some of the principle native platforms for MPSoC simulation and key performance estimation techniques. We also focused on binary translation and compiled simulation techniques to evaluate their feasibility for modeling VLIW machines. Simulation techniques based on hardware/software interfaces are promising and provide realistic co-simulation environments. We reviewed different interface layers and their typical use-cases in native simulation platforms.

Key issues with the existing native platforms include address space differences, memory overlaps, requirement for dynamic linking in software and modifications to the virtual platform components. All of the native techniques require access to software sources and frequently exploit retargetable compilers for modeling target- specific software performance. Performance estimation techniques for the RISC machines are quite mature, providing reasonably accurate results but techniques for VLIW software are scarce.

In the following chapters, we describe the principle contributions of this thesis including a native MPSoC simulation platform based on *Hardware-Assisted Virtualization* (**HAV**) technology and a *Static Binary Translation* (**SBT**) technique for modeling VLIW software on top of the same native platform, with instruction level accuracy.

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

5 Native MPSoC Simulation Platform Using Hardware-Assisted Virtualization

Virtualization of physical computing resources is a well-known concept [Gol74, Cre81] and provides means for sharing these resources to improve system utilization. It is similar to abstraction but details of underlying resources are not necessarily hidden from the software, as it deals with the creation of logical structures that operate just like the real physical machine. In a non-virtualized system only a single operating system is in control of the hardware resources, whereas in a virtualized environment a new software layer is introduced, known as *Virtual Machine Monitor* (**VMM**) or *Hypervisor*, which controls and arbitrates accesses to the platform resources. This enables the use of multiple operating systems on top of a single hardware machine, commonly known as *guests* of the VMM. The VMM presents a set of virtual platform interfaces that constitute a *Virtual Machine* (**VM**) to each of the guest operating system, making them believe they have full control over the "physical" machine. The term *host* is commonly used to refer to the execution context of the VMM or host the Operating System.

Virtualization technologies are broadly categorized into two types *i.e.* paravirtualization and full virtualization. Paravirtualization technologies are transparent to the software applications but not to the guest operating system, as it must be modified to use the API defined by VMM in order to run in the virtual machine. Xen [BDF+03] uses paravirtualization but preserves binary compatibility for user-space applications. Full virtualization requires no modifications to the guest operating system, providing illusion of a complete physical system, including processors, memory and I/O devices. *Hardware Virtual Machine* (**HVM**) is the full virtualization mode supported by Xen and requires hardware support, as introduced by AMD Pacifica [AMD05] and Intel Virtualization Technology (VT) [UNR+05].

The *Hardware-Assisted Virtualization* (**HAV**) technology provides support for full virtualization and solves some of the key challenges to virtualization of IA-32 and Itanium architectures. These include ring deprivileging (causing ring aliasing and address space compression), non-faulting accesses to privileged state, interrupt virtualization and frequent accesses to the privileged resources [NSL+06]. This chapter presents our first contribution

and describes the efficient resolution of target *vs*. host address spaces problem, using the HAV technology provided by state-of-the-art x86 processors, as discussed in Chapter 3. We demonstrate the use of HAV in the context of event-driven native simulation environments for modeling MPSoC architectures.

5.1 Hardware-Assisted Virtualization (HAV)

The HAV technology provides hardware assists for overcoming the difficulties associated with the virtualization of IA-based systems. Principle architectures that support HAV include Intel VT-x (x86), VT-i (IA-64) and AMD-V, previously known as AMD Secure Virtual Machine (SVM). Recently other platforms including PowerPC, ARM Cortex A15 and SPARC have also introduced support for hardware based virtualization.

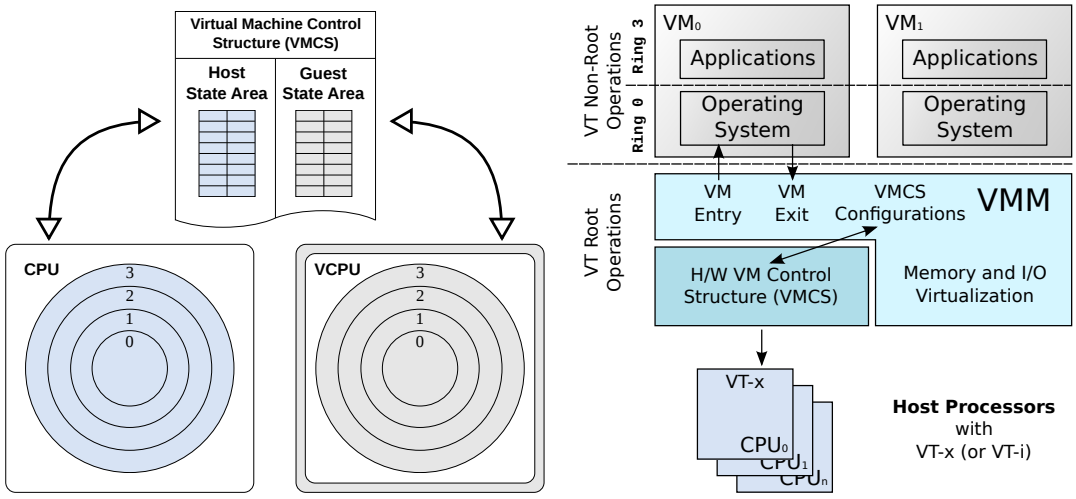
Intel architectures provide a protection mechanism based on the concept of privilege levels. Privilege levels are defined using a 2-bit field resulting in 4 levels, where 0 and 3 define the most and the least privileged levels, respectively. Privilege levels are also used for controlling access to address-space, by employing them in segmentation (IA-32 only) and paging structures. In the virtualization context, a host operating system cannot allow a guest system to run at the same privilege level as itself, so it has to either *deprivilege* its guest or find other means to accomplish this objective. Other significant difficulties include address space translation and interrupt virtualization, which become performance bottlenecks in the existing software based virtualization technologies. The HAV technology has been introduced to target these complex problems by providing dedicated hardware [UNR+05, NSL+06] and supports the following key features:

- **New Guest Operating Mode**: This mode provides a new execution context of the host machine in which the address space can be fully customized. Guest software executes in *non-root operation* mode whereas the host software and VMM execute in *root operation* mode. The guest mode provides all four privilege levels except that the VMM can configure for certain instruction executions and register accesses to be trapped.
- **Hardware-based State Switch**: Hardware support for *atomic* switching between guest and host modes and vice versa., in a complete and efficient manner, has been introduced. The hardware switches the control registers, the segment registers and the instruction pointer so that both address space switching and control transfer are performed atomically. Two new types of transitions are defined: a transition from root operation to non-root operation mode known as VM Entry, and a transition from non-root operation to root operation mode is called a VM Exit.
- **Guest Mode Exit Reason Reporting**: Each time the guest software quits guest mode, it reports the exit reason to VMM, which uses this information to take an appropriate action. Exceptions and interrupts are examples of reasons for leaving the guest mode.

5.1.1 Processor Virtualization

In traditional software based virtualization techniques, some guest software instructions are not allowed to directly execute on the host processor. These are known as *sensitive* instructions, as they can interfere with the processor state, affecting the host operating system and VMM behavior. The guest operating system is executed in unprivileged mode

so that execution of sensitive instructions results in traps and can be emulated by the VMM. This strategy works fine, given that all of the sensitive instructions are privileged[1] as well, thus meeting the Popek and Goldberg [PG74] requirements on virtualizable architectures. The x86 architecture contains around 17 instructions that are sensitive but unprivileged, creating a difficult situations for software only virtualization techniques. For example the IA-32 registers GDTR, LDTR, IDTR and TR contain pointers to data structures that control processor operations. Software can write to these registers using `LGDT`, `LLDT`, `LIDT` and `LTR` instructions at privilege level 0 only. However, it can read from the same registers using `SGDT`, `SLDT`, `SIDT` and `STR` instructions at any privilege level. This means that the guest OS can deduce existence of a virtual machine and the fact that it does not have full control over the CPU, in situations when the VMM maintains unexpected values in these registers. See Appendix A for more details on privileged and sensitive instructions.



(a) Host *vs*. Guest Mode in HAV (b) VMCS, VT Root and Non-Root Operations

Figure 5.1: Guest *vs*. Host Modes in *Hardware-Assisted Virtualization* (HAV) (Intel Processors)

HAV technology solves this problem by providing a dedicated guest execution mode, also known as *Virtual Machine eXtensions* (**VMX**), without constraining the software by privilege levels. Figure 5.1(a) shows the guest and host modes, where the host and guest softwares have access to all four privilege levels. The guest and host states are maintained in the host memory using structures designated as *Virtual Machine Control Structure* (**VMCS**) or *Virtual Machine Control Block* (**VMCB**) by Intel and AMD, respectively (Figure 5.1(b)). One such structure is allocated by the VMM for every *Virtual CPU* (**VCPU**) and stores the processor state of host and guest modes in Host and Guest State Areas. This information includes necessary configurations such as Segment Registers, CR3, IDTR and fields specifying the interruptibility state of the guest processor. For performance reasons, the general purpose registers are not saved in the guest/host state areas and this responsibility is delegated to the VMM. The VMCS controls interrupts and exceptions; thus, it can configure guests to cause VM Exit conditionally using the VM-execution control fields [UNR+05]. The VMCS also includes VCPU state fields that are not represented in any software accessible register. Every VM Entry and VM Exit loads and saves these fields, in order to preserve the VCPU state while the VMM is

[1] A privileged instruction, when executed in unprivileged mode results in a trap (General Protection Fault).

running.

The VM entries save the host mode processor state to the host state area, and load guest mode processor state from the guest state area. VM exits perform the reverse actions and additionally store information about the cause of VM Exit in the VMCS *i.e.* the exit reason and the exit qualification. The VMCS also specifies the conditions that control when the VM entries and exits take place. For example, controls that specify interrupt virtualization, including external interrupts and interrupt window exiting, to control the guest software behavior. Moreover, the VMM can control guest paging operations by protecting certain bits using register masks, and guest writes to these bits cause VM exits. Additionally, bitmaps can be set for exceptions, I/O operations and *Model Specific Registers* (**MSRs**).

5.1.2 Memory Virtualization

Every VMX transition between the host and guest can switch address spaces because the CR3 register is included in the guest and host state areas of VMCS. This feature allows the guest software to use its full address space, which can be different from that of the host and VMM. All VMX transitions are controlled by VMCS and it resides in the physical address space of host machine instead of the linear address space of guest system. This eliminates the need to search the VMCS in the linear address space of guest, as it can be different from the linear address space of VMM [UNR+05]. Guest address space is part of the user address space on the host machine, which defines the guest physical to host virtual address translations. The VMM provides the following types of address translations:

- *Guest Physical Address* (**GPA**) to *Host Physical Address* (**HPA**) translations when guest paging is disabled and the software uses physical addresses directly.
- *Guest Virtual Address* (**GVA**) to *Guest Physical Address* (**GPA**) to *Host Physical Address* (**HPA**) translations when guest paging is enabled.
- *Nested Guest Virtual Address* (**NGVA**) to *Nested Guest Physical Address* (**NGPA**) to *Guest Physical Address* (**GPA**) to *Host Physical Address* (**HPA**) translations when guest launches a guest of its own, and paging is enabled in both the guest and the nested guest.

The above-mentioned translations are provided using either *Shadow Page Tables* (**SPT**) or by exploiting the two-dimensional paging hardware support known as *Extended Page Tables* (**EPT**) or *Rapid Virtualization Indexing* (**RVI**)[2] by Intel and AMD, respectively. In essence, the VMM is responsible for exposing a standard x86 *Memory Management Unit* (**MMU**) to the guest software, while translating guest virtual or physical addresses to the host physical addresses, using either of these technologies.

5.1.2.1 Shadow Page Tables

Memory virtualization using *Shadow Page Tables* (**SPT**) is provided by HAV-based VMMs by intercepting all paging operations of the guest software including page faults (#PF exceptions), page invalidations (INVLPG) and accesses to the emulated guest control registers (CR0, CR2, CR3, and CR4). Essentially, all attempts from the guest software to access address-translation hardware are trapped by the VMM and emulated. The VMM must ensure that

[2] Formerly known as Nested Page Tables (NPT).

accesses to page directories and page tables get trapped and this is accomplished using the conventional page-based protection mechanisms. Additionally, traps for system registers accesses are configured using the VMCS, which specify what actions the guest should perform and when to do a VM Exit and give control to VMM. This usually happens when any of the system register is accessed by the guest. Once VMM takes control, it updates its SPTs, which provide GPA (or GVA or NGVA) to HPA translations.

Updates to SPTs are also triggered when the host modifies its own translations *i.e.* either GPA to HVA or HVA to HPA. As the host OS maintains full control over its memory, it can even swap-out some of the pages assigned to guest at anytime, typically when memory pressure builds up in the system and the memory shrinker is activated. Such events also trigger updates to the SPT entries maintained by the VMM.

Shadow page tables occupy significant amount of host memory, which is directly proportional to the number of VMM guests. Secondly, the synchronization operations between guest and shadow page tables cost valuable CPU time and degrade guest performance. Thirdly, multi-tasking workloads suffer greatly, as on each context switch the shadow page tables have to be invalidated and rebuilt for the new task. Reconstruction of SPTs is much more expensive than refilling of the *Translation Lookaside Buffer* (**TLB**). To reduce these overheads, caching techniques for SPTs across context switches have been introduced [KKL+07]. Figure 5.2(a) shows the general idea behind SPTs. Kindly refer to Appendix B.1 and Chapter#31 in [Int11] for further details.

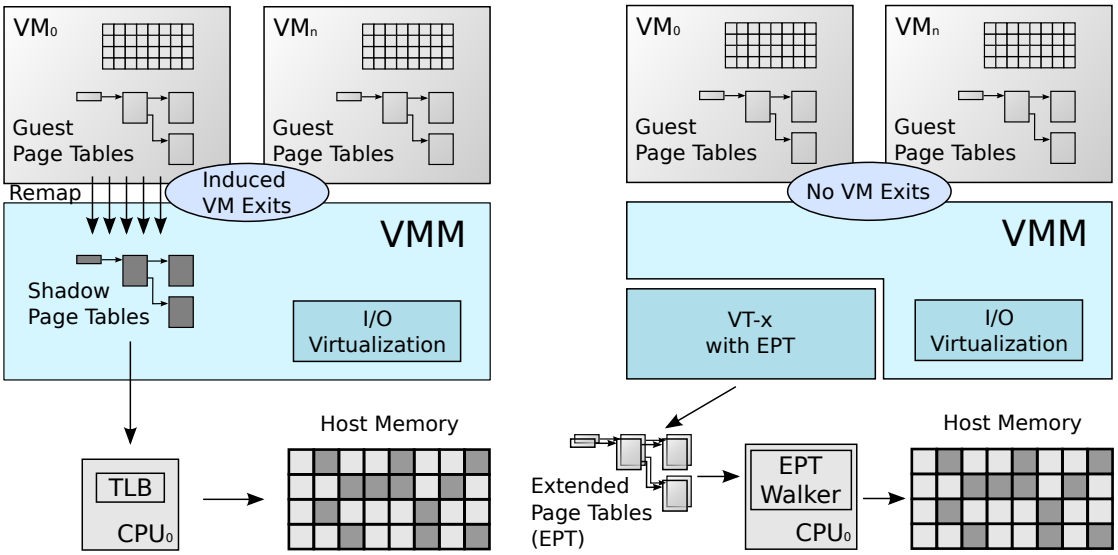
5.1.2.2 Extended Page Tables

When two-dimensional paging support is enabled in HAV, the guest physical addresses as used in non-root operation mode are translated by traversing through a set of EPT (or RVI) paging structures. These structures are implemented in hardware, and an EPT page walker translates the GPAs to final HPAs that are used to access the host memory. The guest operating system is allowed to freely modify its page directories (referenced by CR3 register) and page tables, as the guest paging structures become independent of the VMM and guests can directly handle page faults. This feature eliminates the need to quit the guest mode for paging operations, as is the case for shadow page tables, resulting in improved guest performance and reduced memory consumption.

EPT paging structures are used to translate all guest memory operations that *actually* access host physical memory, without requiring the guest OS to do a VM Exit. For example, when guest maintains its own paging structures and uses them to translate guest virtual addresses (GVAs) into guest physical addresses (GPAs), the EPT paging structures are involved at each translation stage. During guest translation steps, the page directory entries and page table entries go through EPT to access host memory and the resulting GPA is translated using EPT to determine the final HPA.

The original VMX architecture requires flushing of TLBs and paging-structure caches on each VMX transition (VM entries and VM exits). A new feature known as *Virtual Processor Identifiers* (**VPIDs**) (named as *Tagged* TLBs by AMD) allows to associate logical processor identifiers with TLB-caches and eventually cache translations for multiple linear address spaces can be maintained. Kindly refer to Appendix B.2 and Chapter#28 in [Int11] for more details on two-dimensional paging.

As discussed in Section 3.5.3, the key issue in native simulation is the transparent handling of address translation in order for a target application to access the simulated



(a) Using Shadow Page Tables (SPT)

(b) Using Extended Page Tables (EPT)

Figure 5.2: Memory Virtualization Support in *Hardware-Assisted Virtualization* (HAV)

memory and peripheral components, and symmetrically for a simulated processor or DMA to access target allocated buffers. Memory virtualization capabilities provided by the HAV technology can very conveniently help to resolve the target *vs*. host memory addressing problem in native MPSoC simulation platforms, as elaborated in the up-coming text.

5.1.3 I/O Virtualization/Emulation

The HAV technology provides processor and memory virtualization capabilities, as discussed in the preceding sections. For I/O device emulation HAV technology relies on the VMM to provide either in-kernel support, or forward such requests to the user-space device emulators such as *Quick EMUlator* (**QEMU**). Most of the host-specific devices such as *Programmable Interrupt Controller* (**PIC**), *Local APIC* (**LAPIC**), *Programmable Interval Timer* (**PIT**) and *I/O APIC* (**IOAPIC**) are emulated by th VMM in kernel mode.

More recently the concept of *Input/Output Memory Management Unit* (**IOMMU**) has been introduced by Intel and AMD, named Intel VT-d (Direct I/O) and AMD-Vi (I/O Virtualization), respectively. An IOMMU enables guest virtual machines to directly use peripheral devices, such as Ethernet, accelerated graphic cards and hard-drive controllers through DMA and interrupt remapping techniques.

This chapter will primarily focus on the use of processor and memory virtualization capabilities for the purpose of native simulation. The I/O emulation capabilities provided by VMM, such as *Local APIC* (**LAPIC**) for multiprocessor modeling, will be used and user-space device models will be provided by a SystemC-based simulation platform.

5.2 Native Simulation Using Hardware-Assisted Virtualization

We solve the target *vs*. host address spaces problem in native simulation by introducing a transparent and hardware-based address translation layer provided by the HAV technology. This translation scheme is different from the *Dynamic Binary Translation* (**DBT**) based ISSes, which invoke software based address translation functions for each memory access, thus degrading the simulation performance.

The following sections provide detailed description of how we solve the address spaces problem by integrating a VMM into an event driven simulation environment, for the sake of concreteness, we integrate the open source Linux-based *Kernel Virtual Machine* (**KVM**) [KKL+07] into SystemC environment. Our contribution is not to create a fundamentally new HAV-based virtual machine monitor, but to use an existing one and integrate it into an event-driven SoC simulation environment to solve the problems mentioned in Chapter 3 (Sections 3.5.3 and 3.5.4).

KVM is composed of two parts, a Linux kernel mode driver (the VMM) and a user-space library that bridges the gap between the kernel driver and user-space software such as SystemC based virtual platforms. We will provide details on how we can simulate an entire software stack including an operating system and multi-threaded software applications on multiple processing elements.

5.2.1 Native Processing Units

The *Native Processing Units* (**NPUs**) are based on the SystemC module concept and form the basis of our simulation framework. Each NPU models a native processor and provides the interface between hardware SystemC components and KVM. For interfacing with KVM, it uses the KVM user-space library, which exports key functions for virtual machine manipulation. The KVM kernel module exposes the host processor ISA to the software stack *e.g*. the x86 ISA in our case. The software stack, including the embedded OS, the HAL layer and the application are compiled to the host binary format and executed on top of the VMM.

Each native processor includes a SystemC thread to model the processor and components for interacting with the KVM user-space library, such as callback functions and features to exploit the KVM library interface. The native simulation platform requests certain services from the KVM, such as creation of a new *Virtual Machine* (**VM**) including one or more *Virtual CPU* (**VCPU**), initialization of the guest memory space and launching the software execution. Each of these requests is sent to the KVM kernel module across the user/kernel mode frontier using the `ioctl()` mechanism provided by the Linux kernel. Each `ioctl` returns a value to indicate success or failure of the requested service.

The native processors install certain callback functions into the KVM user-space library that will be executed whenever the simulated software stack has some events for the SystemC platform, or requires certain services neither provided the VMM nor by the user-space library. These events include I/O accesses, exceptions, profiling and timing annotation requests, which are transfered to the native processor that will either handle them locally or forward them to the SystemC platform. On the hardware side, the native processor provides the SystemC interfaces, including the TLM ports for transactions on the communication network. Figure 5.3 provides a high-level view of native processing units and their relationship to KVM. Table 5.1 summarizes the basic KVM APIs to create a new virtual machine, to assign user-space memory to it, and to create VCPUs. The current maximum number of VCPUs limit

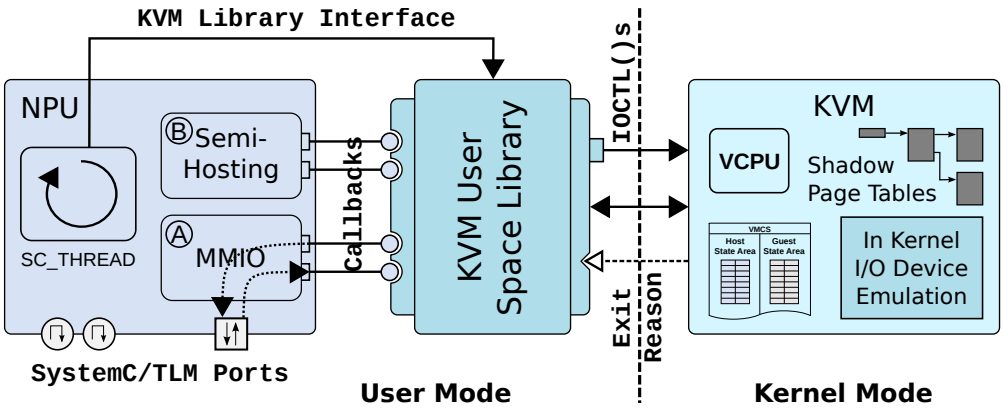


Figure 5.3: Native Processing Units and their interfacing with KVM using the User-space Library

stands at 254 VCPUs per virtual machine in KVM on x86 hosts.

KVM API	Description
KVM_CREATE_VM	Create a new Virtual Machine, without any VCPUs and Memory.
KVM_CREATE_VCPU	Create a new VCPU in an existing Virtual Machine.
KVM_SET_USER_MEMORY_REGION	Create or modify a guest memory slot (Allocated in User-space).
KVM_[GET|SET]_[REGS|SREGS]	Get or Set General Purpose or System Registers of a VCPU.
KVM_RUN	Start or Resume Execution (Returns with an Exit Reason).

Table 5.1: Basic KVM APIs for Virtual Machine Initialization and Execution

The KVM_RUN API is the most important one and is used to start or resume execution on a particular VCPU in a virtual machine. Each execution of this API results in host to guest mode switch and the guest software is executed until it reaches an instruction that cannot be executed in guest mode, or requires assistance from the host machine. The guest mode quits and gives control to VMM, which restores the host state and tries to fulfill the guest's requirement within kernel mode. Often due to a page fault, an exception, an I/O request, or the need to emulate an instruction, as the guest is not allowed to execute certain sensitive instructions. For this purpose the VMM has access to exit reason and exit qualification fields saved by the HAV support. The exit reason provides the general cause of guest exit to host mode, while the exit qualification provides further details in certain cases, such as page faults and debug exceptions. For example, in case of a guest page fault, the exit qualification contains the linear address that caused the page fault. Figure 5.4 shows the overall execution flow of native simulation using KVM. A very similar exit reason based switching structure is used in KVM library for forwarding requests to appropriate user-space components, as given in Listing 5.1.

5.2.2 Host Dependent Hardware Abstraction Layer

Native simulation platforms face two types of dependencies, mainly resulting from the native compilation of software. The differences in ISA of host and target processors, as

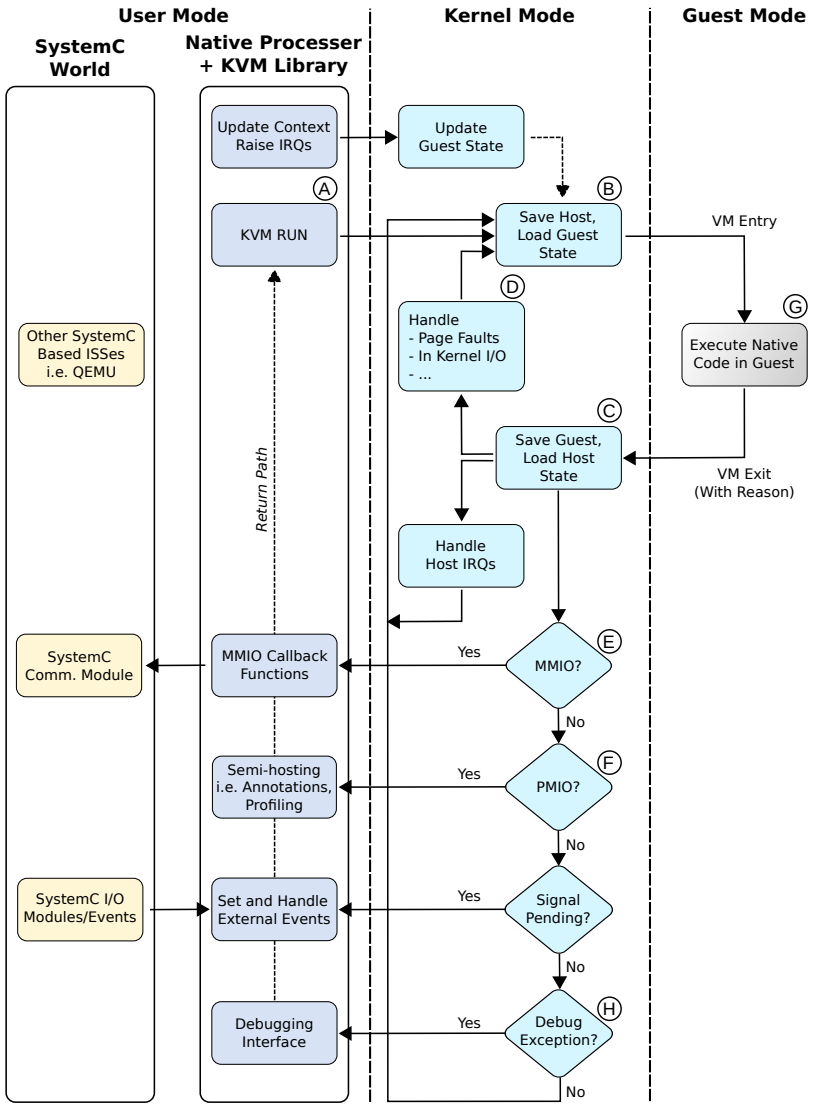


Figure 5.4: Execution Flow in Hardware-Assisted Virtualization Based Native Simulation

well as specific details of hardware modules, need to be addressed in the first place. These differences are resolved by explicitly using the HAL API for *all* interactions between software and hardware components [YBB+03, GGP08], resulting in a *hardware-independent* software stack, except for the HAL layer implementation. As a result, our approach is similar to paravirtualization. Second source of dependencies arise due to the memory representations in hardware and software components, as discussed in Section 3.5.3 and we address this issue in the following section (*c.f.* Section 5.2.4).

HAL layer has to bridge the gap between two worlds; firstly it has to provide an interface that is specific to the operating system running on top of it, and secondly it has to implement the HAL interface in terms of the underlying platform. We have implemented our HAV based approach for DNA-OS [GP09], so the HAL layer respects the interface expected by DNA-OS. As we base our solution on top of the HAV technology, which exports host-specific ISA to its guests so the HAL layer is implemented using the host ISA (x86 in our case). Using the HAV

Listing 5.1 Execution Flow in KVM user-space library

```
1  int kvm_cpu_start(struct kvm_cpu *vcpu)
2  {
3      /* Local Variables */
4      reset_vcpu(vcpu);
5      ...
6      while (vcpu->is_running)
7      {
8          ioctl(vcpu->fd, KVM_RUN, 0);
9
10         switch (vcpu->exit_reason)
11         {
12             case KVM_EXIT_MMIO:
13                 /* Handle MMIO request; forward to SystemC */
14                 break;
15             case KVM_EXIT_IO:
16                 /* Handle I/O request; used for Semi-hosting */
17                 break;
18             case KVM_EXIT_INTR:
19                 /* Handle Signal */
20                 break;
21             case KVM_EXIT_DEBUG:
22                 /* Handle debug exception */
23                 break;
24             case KVM_EXIT_SHUTDOWN:
25                 /* Stop VCPU */
26                 return 0;
27             ...
28             default:
29                 goto panic_kvm;
30     }
31 panic_kvm:
32     return 1;
33 }
```

technology, all interactions between software and hardware components take place at the host ISA level, as was discussed in Section 4.1.2 (Figure 4.4(d)). The HAL layer provides APIs for managing process contexts, synchronization primitives, endianness, memory and I/O accesses and interrupt management. Chapter 7 will provide more details on these APIs.

Figure 5.5 shows the fact that all of the software layers above HAL are independent of the underlying platform, and only HAL is specific to the host machine architecture. As a concrete example, we give the implementation of `CPU_TEST_AND_SET` HAL API function in Listing 5.2, in comparison with the Listings 3.2 and 3.3, which show the implementations for classic native and ARM processor, respectively.

Once the initial design space exploration is finished for a given SoC, the same software stack can be re-compiled for the target architecture excluding the HAL layer and replacing it with a target specific implementation. This enables the validation of a significant amount of software for the target architecture, including the operating system and all software layers above it. As the context initialization and switching functions are included in the HAL layer, modeling of dynamic task creation and thread migration within SMP architectures is possible – a key requirement in recent SoCs.

The native software executes on top of a virtual machine and the VMM controls the resources accessed by the guest software. Moreover, the simulated software does not know the existence of a virtual machine, and executes in isolation. Thus, software execution is very similar to that on the final target architecture, as host resources are *invisible* to the guest

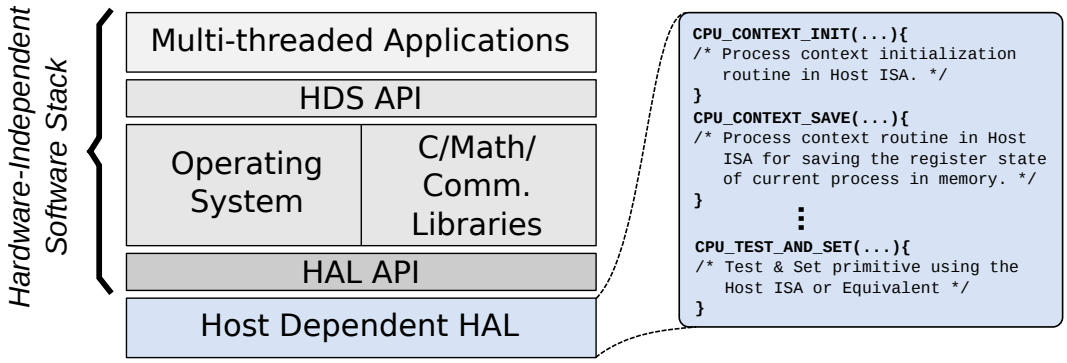


Figure 5.5: Host Dependent *Hardware Abstraction Layer* (HAL)

Listing 5.2 HAL Function `CPU_TEST_AND_SET()` for HAV-based Native Host (x86)

```
1  long int CPU_TEST_AND_SET (volatile long int * spinlock)
2  {
3      long int oldval = 0;     /* Previous value to compare with. */
4      long int newval = 1;     /* New value for taking the spinlock. */
5      unsigned char flag;      /* flag: 0 -> Success, 1-> Failed to lock. */
6
7      __asm__ volatile (
8          "lock cmpxchgl %1, %2\n"
9          "setne %%al"
10              : "=a" (flag)    /* Non-zero means lock is taken by someone else */
11              : "r" (newval), "m" (*spinlock), "0" (oldval)
12              : "memory");
13
14     return ((long int) flag);     /* Return 0 means OK */
15 }
```

Key Point 5.1 Software Validation

All software layers above the HAL, including operating system, libraries and application remain un-modified, thus validating a significant amount of software. In order to re-use the same software stack on a different target platform, only the HAL has to be replaced with a target-specific version and re-validated.

software. This is in contrast to approaches where the simulated software is allowed to directly use the standard C API functions available on the host machine, resulting in inaccuracies as the software executes outside of the simulation platform.

5.2.3 Using Hardware Abstraction Layer as a Synchronization Point

In the absence of annotations, software can only synchronize with hardware models when an I/O takes place, or an external event occurs. To increase the synchronization granularity, we can use the HAL layer as a synchronization point, in addition to its normal role of providing hardware abstraction. This idea (proposed in [Ger09]) ensures that hardware simulation kernel can update the simulated time and avoid deadlocks. For example, if a software thread uses a loop to wait for a device register's value to change, it requires that the SystemC scheduler gets the control and advances the simulated time. Once the simulated time changes, SystemC can executes a different software or hardware process, which might change the

register contents. To give control back to SystemC kernel, the `wait()` function must be invoked, otherwise the simulation cannot progress. Using this idea, each HAL API calls the SystemC `wait()` function for consuming a certain amount of time and the above-mentioned deadlock case can be avoided.

The principle difficulty with this idea lies in the total absence of information about the time value that should be passed to the SystemC `wait()` function *i.e.* the value of synchronization time variable T_{SYNCH}. A key factor to this imprecision comes from the fact that software executes in *zero-time w.r.t.* to the simulated hardware timings. The time value has no significance when we use an un-timed hardware model. However, we cannot attribute any arbitrary value to T_{SYNCH}, as it can have a non-negligible impact on the application behavior and this time value indirectly translates to the processor performance. The timing aspect will cause problems when the software stack starts using the notion of time, for example when scheduling different processes, or in case of real-time systems. To ensure the functional correctness of an application, the *load condition* given in Equation (5.2.1) must be respected when choosing a value for the synchronization time.

$$\sum_{i \in E} \frac{C_i}{T_i} \leq 1 \tag{5.2.1}$$

where E is the set of external event sources, C_i is the cost of handling event i and T_i represents the time separation between two occurrences of the i events. To make the concept more clear, we present an example in Figure 5.6 where a single processor handles events from three different sources: $Event_0$, $Event_1$ and $Event_2$ occur every `10`, `5` and `11` time units and are handled in `2`, `1` and `3` time units respectively. Calculating the load condition gives us:

$$\frac{C_0}{T_0} + \frac{C_1}{T_1} + \frac{C_2}{T_2} \implies \frac{2}{10} + \frac{1}{5} + \frac{3}{11} \approx 0.67 \leq 1$$

The result indicates that under the given conditions, system will continue to function properly, which is also evident from the processor timings in the above example (Figure 5.6). In summary, the cost of handling all external events must be less than or equal to unity, to make sure that the processor does not miss any events.

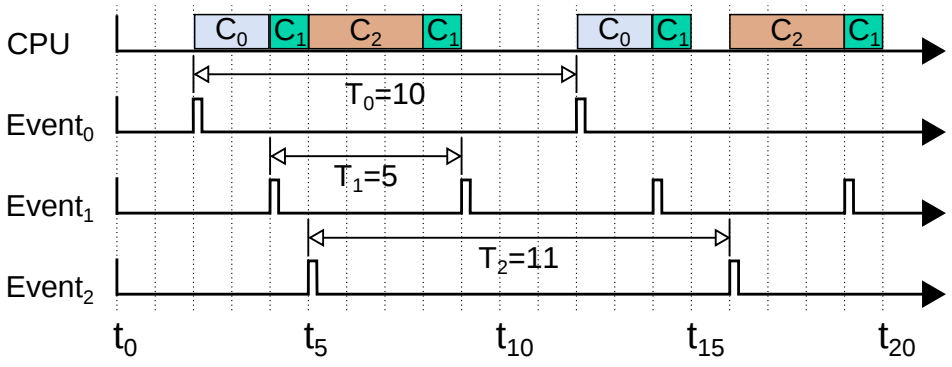


Figure 5.6: Multiple External Event Sources, their Cost and Load Condition

On a real processor the value of C_i is the actual processing time for the event i, and this time could be estimated using *Worst Case Execution Time* (WCET) techniques, or measured on

a cycle accurate or real platform. In case of native simulation, the granularity of software execution *w.r.t.* hardware model is the HAL API calls, so the cost of handling event i becomes the number of HAL API calls invoked during the handling of event i multiplied by the value of time function T_{SYNCH}. So the Equation (5.2.1) can be re-written as:

$$\sum_{i \in E} \frac{HAL\_API\_CALLS_i \times T_{SYNCH}}{T_i} \leq 1 \tag{5.2.2}$$

Ideally, we should use the *actual* time spent by the target processor for executing such events, but it is a difficult problem. Moreover, we do not have any mathematical model that could approximate this time value. Two consecutive HAL function calls do not give us any information about the number of target instructions executed in-between them, and the same limitation holds for the total number of HAL API calls and the number of instructions executed in a given application. Nevertheless, the above equation allows us calculate an approximate value for T_{SYNCH}, for use in un-timed software applications, and to properly handle the hardware events such as interrupts, as shown in Figure 5.6. The choice of a value for T_{SYNCH} simply becomes a constant that respects the Equation (5.2.3).

$$0 < T_{SYNCH} \leq \frac{1}{\sum_{i \in E} \frac{HAL\_API\_CALLS_i}{T_i}} \tag{5.2.3}$$

The number of $HAL\_API\_CALLS_i$ can vary for different executions of the same event handlers at different times, mostly due to different architectural states of the system such as cache memories, communication congestion, *etc.* As a consequence, the value of $HAL\_API\_CALLS_i$ in Equation (5.2.3) should represent the maximum possible number of HAL API calls for handling an event i, in order to ensure that the system will always respect the load condition. Lets suppose that $Event_0$, $Event_1$ and $Event_2$ handlers require a maximum of $H_0 = 1000$, $H_1 = 1200$ and $H_2 = 1500$ HAL API calls, respectively. If all time units are expressed in milliseconds *i.e.* $T_0 = 10ms$, $T_1 = 5ms$ and $T_2 = 11ms$, Equation (5.2.3) gives us the following result:

$$\frac{1}{\frac{1000}{10\times10^{-3}} + \frac{1200}{5\times10^{-3}} + \frac{1500}{11\times10^{-3}}} \implies 0 < T_{SYNCH} \leq 54.34ns$$

The result indicates that we can attribute any non-zero value below $54ns$ to T_{SYNCH}, and the processor will be able to handle all external events. We will refer readers to [Ger09] for further explanation and experimental results of the HAL layer based synchronization concept.

As our proposed solution uses a HAL layer that executes within the virtual machine environment, a direct call to the SystemC `wait()` function is impossible. We can nevertheless support this idea by using a guest mode stub that takes each of the synchronize function calls and transfers them to the SystemC simulator using *Port-Mapped I/O* (**PMIO**) support on the host machine, as shown in Figure 5.7. We must emphasize that the use HAL layer as a synchronization point is only to ensure functional correctness and proper handling of external events. This concept does not relate, in anyway, to the software performance estimation techniques, and is merely a mean to ensure a working solution in case of timed hardware and software systems.

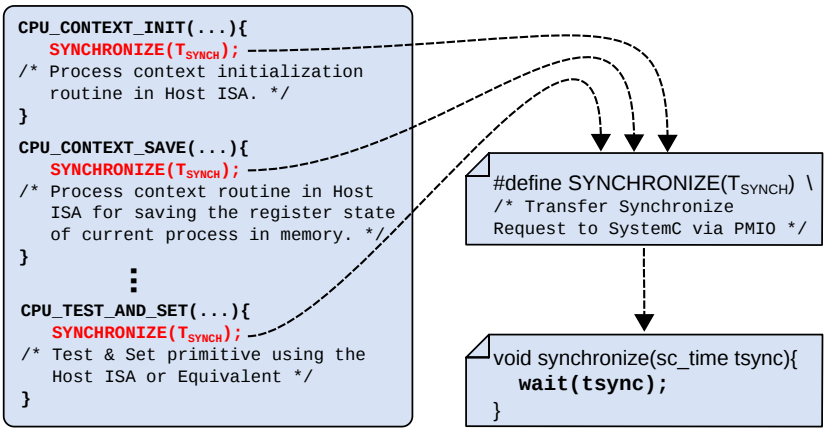


Figure 5.7: Hardware Abstraction Layer with Synchronizations

5.2.4 Memory and I/O Address Space Accesses

Guest software believes that it has access to a full zero-based memory address space and the VMM preserves this illusion either using shadow pages, or by employing hardware support as discussed in Section 5.1.2. The key to memory virtualization is the translation of guest addresses (GVA or GPA) directly to host physical addresses (HPA). In case of shadow pages, guest accesses to un-mapped memory result in traps to kernel mode where the VMM walks the guest page tables (if any) to determine the guest "physical" addresses. The VMM then performs a translation of GPA to HPA and installs a shadow page entry containing GPA (or GVA) to HPA translation and restarts the execution of faulting instruction. In case of hardware support the translation of GPA to HPA is delegated to the dedicated hardware page tables.

The Figure 5.8 is borrowed from [KKL+07] and modified to focus on how both the memory and I/O address spaces from the target system are mapped into the user address space, making them accessible to SystemC memory models as well. For the simulated software stack that knows only about the *target* address space, the "physical" memory addresses used by the target binary are the virtual ones mapped by the KVM Linux kernel module to a series of real physical memory pages of the host platform (small boxes). All SystemC modules can access these physical pages by using another MMU mapping which is also maintained by the KVM kernel driver. As access to these pages is fully transparent *i.e.* a program running in guest mode will stay in guest mode, it leads to an optimal bi-directional data sharing between the target binaries and the SystemC environment.

In the multi-core architectures, this memory can also be shared among multiple cores (using the `mmap()` system call) and SystemC components, which fundamentally breaks the bottleneck of native simulation mentioned in Section 3.5.3. User-space processes such as SystemC can create multiple "physical" memory slots for their guests; thus, modeling of different memory hierarchies is possible. Current limit on the number of memory slots stands at 32 for a given virtual machine in KVM on an x86 host and total memory limit is subject to the actual physical resources of the host machine.

The KVM kernel module provides the `KVM_SET_USER_MEMORY_REGION` API to build a mapping relationship between the user-space allocated memory and share it with the guest. Due to this memory mapping mechanism, the target software stack can stay in guest mode and access the memory without notifying the SystemC memory components. At the same

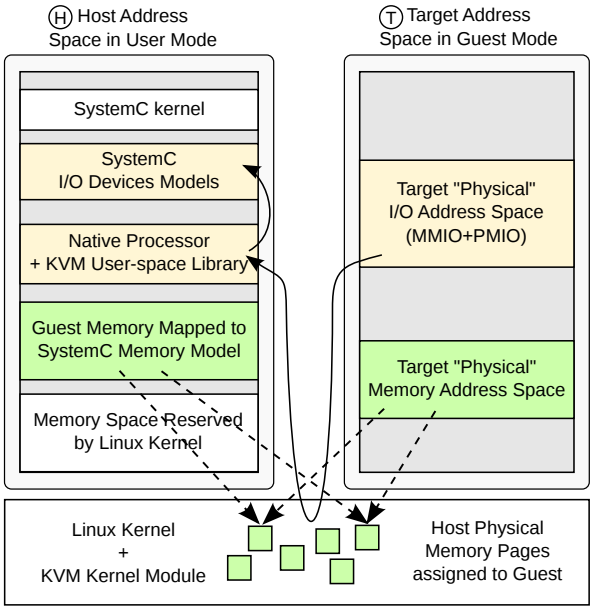


Figure 5.8: Memory and I/O Address Space Accesses

time, this memory is shared among all of the SystemC components and even other ISSes. The details of how to share memories among multiple processors in hybrid simulation platforms will be discussed later (*c.f.* Section 5.5). Listing 5.3 gives a concrete example of how user-space allocated memory can be registered with KVM and shared with the guest software.

Key Point 5.2 MMU Based Operating Systems and Transparent Memory Access

Using HAV approach, all types of operating systems can be simulated, whether they use MMU-based virtual addresses or directly manage the physical address space. Moreover, memory accesses are transparent and software execution remains in guest mode for all memory references, except for page faults and page table manipulations.

5.2.4.1 Memory-Mapped *vs.* Port-Mapped I/O

Besides memory accesses, target embedded software also needs to access I/O devices. *Memory-Mapped I/O* (**MMIO**) and *Port-Mapped I/O* (**PMIO**) are the two well-known methods that a CPU can use to perform I/O accesses. As the x86 architecture supports both methods, theoretically, the simulated target software can access I/O devices with both MMIO and PMIO methods. However, since most embedded processors only support MMIO, such as ARM, we assume that IPs are accessed through MMIO only.

As opposed to the memory accesses, the MMIO cannot be mapped and accessed directly as the behavior of hardware register read/write access is normally different from that of the memory. A hardware register access may trigger some actions from the target hardware component instead of just reading/writing a data value. As all I/O devices of an MPSoC are actually modeled within SystemC, instead of the real hardware, SystemC should get the control and pass it on to the hardware components, whenever an MMIO load/store instruction is executed. This process can be naturally realized on a KVM based virtualization platform.

Listing 5.3 KVM Set User Memory Region [Code snippet taken from KVM Library]

```
1  /*
2   * Note: KVM_SET_USER_MEMORY_REGION assumes that we don't pass overlapping
3   * memory regions to it. Therefore, be careful if you use this function for
4   * registering memory regions for emulating hardware.
5   */
6  int kvm_register_mem(struct kvm *kvm_inst, uint64_t guest_phys, uint64_t size,
7                       void *userspace_addr)
8  {
9      struct kvm_userspace_memory_region mem = (struct kvm_userspace_memory_region)
10     {
11         .slot               = kvm_inst->mem_slots++,
12         .guest_phys_addr    = guest_phys,
13         .memory_size        = size,
14         .userspace_addr     = (unsigned long)userspace_addr,
15     };
16
17     return ioctl(kvm_inst->vm_fd, KVM_SET_USER_MEMORY_REGION, &mem);
18 }
```

Key Point 5.3 Software Coding Constraints and Memory Overlaps

Software can access the full address space including overlapping memory regions that usually conflict with host operating system, in traditional native techniques. Guest software can use any type of memory addressing including hard-coded addresses. If such an address falls within the memory regions mapped by the VMM, it is transparently accessed, otherwise a trap is generated to the VMM and an appropriate action/emulation is performed.

When the simulated target software performs MMIO accesses using an address that does not belong to the memory space, a page fault exception is thrown by the hardware MMU that causes the processor to leave the guest mode and lets the KVM kernel driver handle the exception. The driver transfers the MMIO access to the KVM user-space library, which forwards this access to the native processor, using the callback functions installed during initialization, as shown in Figure 5.4 Ⓔ. With the help of KVM API, the native processor can get the target I/O addresses and launch the normal SystemC I/O operations using the TLM ports and communication components, as shown in Figure 5.3 Ⓐ. Listing 5.4 gives the prototype from KVM library for installing an MMIO callback, which is placed in a red-black tree for quick retrieval.

A native processor installs a callback function for each SystemC component that supports MMIO accesses. Listing 5.5 gives the interface exported by each native processor to the KVM library for use in callback functions. Each time the guest software performs an MMIO request, the MMIO callbacks tree is searched for an address range match, if found the callback is invoked. A read/write request is initiated on the communication component using the functions shown in Listing 5.6, and calls to these functions correspond to TLM transactions.

As the guest software uses target I/O addresses that are exactly the *same* as simulated by the SystemC platform, the communication components can use statically assigned address decoding tables. This particular aspect gives us the freedom to use un-modified hardware models, as opposed to the technique proposed in [Ger09, GHP09], which requires each hardware component to report its *Symbols* and *Mappings* to a dynamic linker, as discussed in Section 3.5.4 (Figure 3.8). Thanks to the static platform decoder address allocation, statically compiled software stack can be used, and does not require run-time linking of platform

component mappings and symbols. Figure 5.9 shows an example where the hardware platform uses compile-time allocated address decoding table for the communication network modeling.

Listing 5.4 MMIO Callback Registration using KVM Library

```
typedef void (*kvm_mmio_callback_t)(struct kvm_vcpu *vcpu, uint64_t addr,
                uint8_t *data, uint32_t len, uint8_t is_write, void *ptr);
bool kvm_register_mmio(struct kvm *kvm_inst, uint64_t phys_addr, uint64_t addr_len,
                        kvm_mmio_callback_t mmio_callback, void *ptr);
```

Listing 5.5 MMIO Call forwarding to SystemC

```
uint64_t systemc_mmio_read (kvm_native_cpu_t *_this, uint64_t addr, int32_t nbytes,
                            int32_t blocking);
void systemc_mmio_write (kvm_native_cpu_t *_this, uint64_t addr, uint8_t *data,
                         int32_t nbytes, int32_t blocking);
```

Listing 5.6 SystemC Read/Write Transactions

```
uint64_t read (uint64_t address, int nbytes, int blocking);
void write (uint64_t address, unsigned char *data, int nbytes, int blocking);
```

Using HAV the address spaces for both memory and MMIO can be exactly the same as the ones on the real target platform, because the address mapping is user-defined and maintained by the KVM user-space library. As opposed to most of the existing native simulation technologies [PV09], our approach is completely independent from the address space of the host machine (generally x86 processor based) and provides as much flexibility as the traditional ISS based solutions. Thus the legacy embedded software can be simulated using our solution, without requiring any modifications (including hard-coded memory and I/O addresses).

Likewise the guest software can use the entire "physical" address-space, including the addresses that are reserved on the host machine for the host operating system. For example, Linux does not allow user-space applications to use virtual addresses above `0xC0000000` on x86 machines, but using the HAV technology we can use such addresses for any purpose, as shown in Figure 5.9. Thus, native simulation platform does not have to deal with memory overlapping issues, as discussed in Section 4.1.2.1. The VMM manages the translation of addresses from user-allocated guest memory address-space to the machine addresses.

As an additional advantage in HAV based approach, the software is allowed to use `while` statements that block on some MMIO-mapped external resources such as a device register. This is different from other native approaches that typically rely on memory-mapped registers where such loop statements will halt the simulation, as SystemC cannot modify the register value unless it gets the execution control. This feature does not imply that memory-mapped external resources are usable in such `while` statements, even in HAV based approach, if they block on some resource held by other processor(s). In such situations an interrupt must be raised from the host platform to the guest software, or an alternative method must be applied to give control back to SystemC (*c.f.* Section 5.4.2).

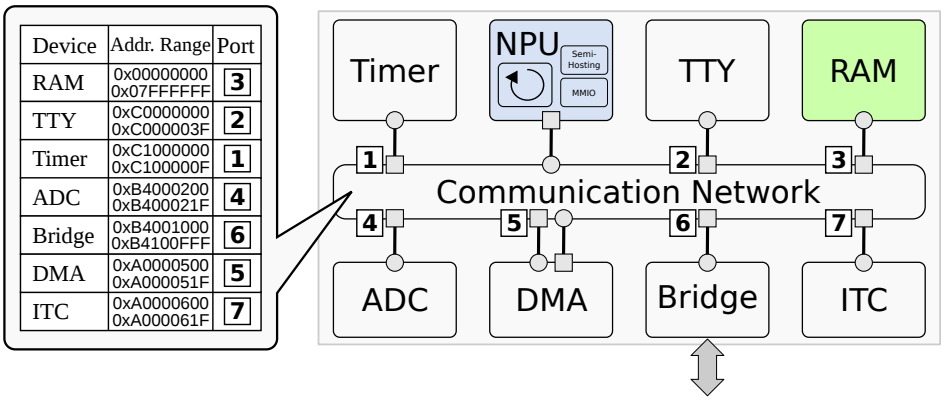


Figure 5.9: Platform Address Decoder with Statically Allocated Addresses

Key Point 5.4 Support for Un-Modified Hardware IP Models

The HAV-based simulation platform executes the guest software in target address space, which matches the address space simulated by hardware IP models. This fundamental property allows the use of un-modified hardware models and avoid dynamic linking of software binaries for address re-mapping.

As a last note, PMIO accesses also cause the VCPU to leave the guest mode, and we use this mechanism to provide a semi-hosting interface for profiling and debug support (*c.f.* Section 5.2.4.2).

5.2.4.2 Semi-hosting and Annotation Support using Port-Mapped I/O

Most of the existing ISSes have the semi-hosting capability to communicate input/output requests from the target software stack to the host machine. The semi-hosting functions, such as functions to measure time on the host machine, are different from the normal I/O operations of the SystemC based SoC platform components. This means that the semi-hosting functions are available even when SystemC components are under development at the early design stage. Hence, with the semi-hosting capability, the native processor can provide extra functional support for debugging, profiling and timing annotations.

As the embedded software stack becomes complex, the semi-hosting function calls provide a simple way to help developers debug their software stacks. Embedded software can print out the debug information directly using the semi-hosting function calls without interacting with the SystemC infrastructure. This feature becomes extremely useful when the embedded software driver or the SystemC hardware terminal devices are not available, or when they have bugs. The proposed solution is capable of debugging the entire software stack using the hardware-assisted capabilities of KVM that can be used for hardware/software co-debugging purposes (*c.f.* Section 5.4.2.3).

The simulation speed of different simulation methods can be compared by profiling the target software stack. The semi-hosting capability is a suitable solution and provides a method to access real clock time of the host machine. As the semi-hosting based profiling functions do not interact with SystemC time management, and usually do not make any transactions on the communication network, they do not influence simulation results. The results of hardware platform simulation remain exactly same as the platform without

profiling information. Listing 5.7 gives an example that can be used by the guest software to write current host time to a log file, using the semi-hosting interface. Working on the same lines, the PMIO accesses can be used for providing annotation support to the guest software. Section 5.3 will provide more details on the annotation technique used in the proposed solution.

Listing 5.7 Profile Request to Write Current Host Time using the Semi-hosting Interface

```
#define CPU_PROFILE_CURRENT_TIME()                                          \
    do{                                                                     \
        __asm__ volatile(                                                   \
            "   mov  %0,%%dx\n\t"                                           \
            "   mov  %1,%%eax\n\t"                                          \
            "   out  %%eax,(%%dx)\n\t"                                      \
            ::"r"((short int) HOSTTIME_BASEPORT),                           \
              "r"((hosttime_request_t) HOSTTIME_CURRENT_TIME)               \
            :"%dx","%eax"                                                   \
        );                                                                  \
    } while(0);
```

For providing the semi-hosting interface and annotation support, the technique used is very similar to the MMIO implementation, with a separate set of callback functions for each service, registered by each native processor using the KVM library (Figure 5.3 Ⓑ). A separate red-black tree for I/O ports is maintained by the KVM library, and each exit from the guest mode goes through this tree to find the appropriate callback function and invokes it (Figure 5.4 Ⓕ and Listing 5.1). All of the semi-hosting functions terminate inside the native processor, without SystemC knowledge except for the annotations calls that execute the SystemC `wait()` function to advance the simulated time.

5.2.4.3 Putting It All Together

So far, most of the basic components of HAV based native simulation framework that solve the target *vs*. host address spaces problem have been discussed. The solution comprises of a Host Dependent HAL layer, KVM kernel module, KVM user-space library and SystemC based native processor models, as shown in Figure 5.10. Two important points need to be revisited; firstly, the memory address space visible to the guest software is the *same* as mapped to the SystemC memory model (connected by a solid line) and is transparently accessible to guest software. Secondly, the I/O accesses initiated by the embedded software go through the kernel module and user-space library to reach the native processor model, which makes the actual read/write transactions on the platform communication network.

Following sections will describe the timing annotation mechanism used in the proposed framework, interrupt support for handling asynchronous external events and support for debugging the software and hardware platform components.

5.3 Timing Annotations in Software

The proposed solution can support any annotation based performance estimation technique, at source level and at basic block level. For the sake of concreteness, we take the annotation scheme proposed in [GHP09] that is based on *Low Level Virtual Machine* (**LLVM**) [LA04].

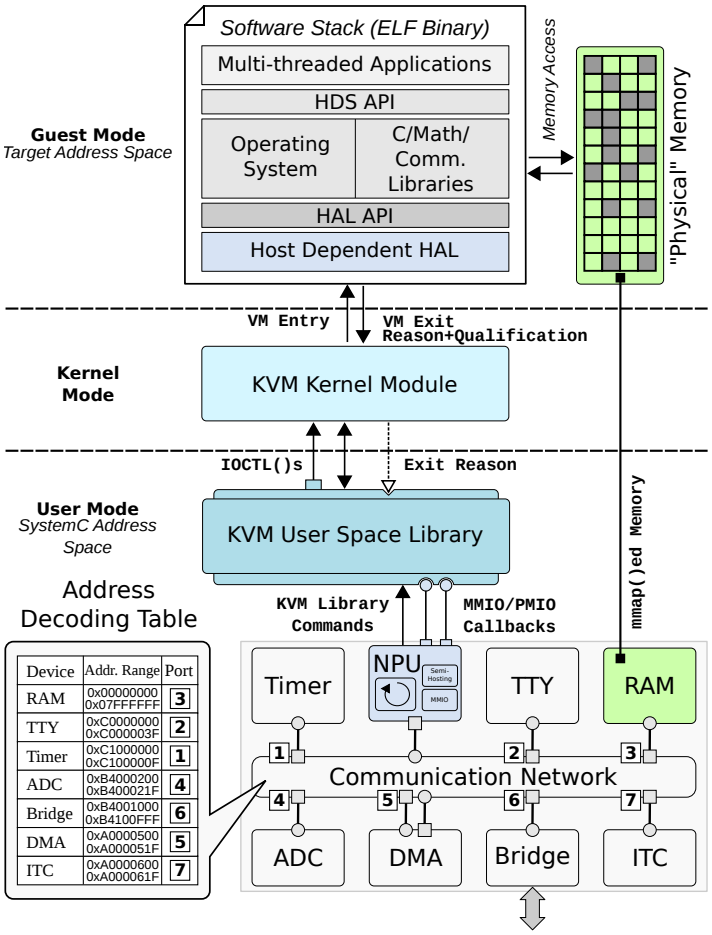


Figure 5.10: Native Processor, KVM Library, KVM and Software Stack in Guest Mode

The key idea behind this approach was presented in Figure 3.9 (Section 3.5.5), where the software is initially compiled for the target machine with target-specific optimizations, but instead of emitting code for the target processor, an annotation pass is introduced. This pass takes the optimized target-specific IR to annotate the target-independent IR, known as *Extended* IR. Figure 5.11 shows the three steps involved in the annotation process for each *Basic Block*. Very similar steps are used for annotating CFG arcs between the basic blocks, for modeling branch penalties where new basic blocks are introduced containing the annotation calls only.

① **Analysis**: A target specific module analyzes instructions in each basic block, by consulting the processor data-sheet and returns an annotation database[3] comprising of instruction count, CPU cycles, memory read/write operation counts and energy estimates, if available.

② **Identification & Storage**: The identification step attaches labels to the annotation databases and stores them in a globally accessible memory section of the object under compilation, for use at run-time.

[3] An annotation *database* is a global data structure containing information for all target basic blocks.

③ **Instrumentation**: Given a target basic block, the annotation pass finds the corresponding target-independent basic block in extended IR and places a call for the `annotate()` function at its start. The `annotate()` function is defined as an external and implemented in HAL of *Execution Units* (**EUs**) in the SystemC platform. The address of the corresponding annotation database is passed as an argument to the `annotate()` function and used at runtime for performance estimation.

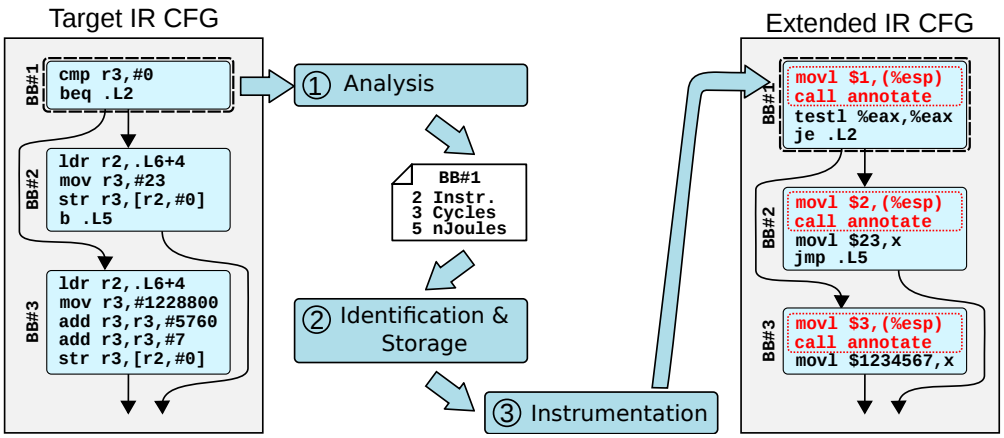


Figure 5.11: Basic Block Annotation Process using LLVM Infrastructure [GHP09]

A fundamental property of the annotation approach is the correspondence between target-specific and target-independent IR CFGs. This property depends on the fact that the two CFGs are *equivalent*, and it is possible to find a corresponding basic block in the extended IR for a given target IR basic block. Such one-to-one correspondence cannot be established between the host and target basic blocks when the code is optimized. In such cases aggregate annotations are introduced, which sum-up the annotation details of multiple target-specific basic blocks onto a single target independent basic block. Similarly, there are cases when a target-specific basic block does not exist in correspondence to a target independent basic block. This is usually due to optimizations and annotations for such basic blocks are skipped. Similar optimizations are also expected from the host backend during host compilation.

At runtime, native basic blocks are executed for functional modeling of the target software, whereas the `annotate()` function calls are used to estimate target processor timings. As the annotation calls are placed at basic block level, it allows to estimate the execution timings of basic blocks *actually* executed at run-time. The native platform proposed in [GHP09] implemented the HAL layer in hardware SystemC models of processors, which also provided the definition of `annotate()` function. In our case, the HAL implementation has been moved to the software stack, compiled as a static binary and executed within the virtual machine environment. Secondly, the SystemC models are hidden from software view, thus a direct function call cannot be placed in the software stack and we have to use a different mechanism for synchronizing software and hardware models.

We use the semi-hosting support for redirecting annotation calls, and provide a local annotation function visible to and compiled with the software stack, as shown in Figure 5.12. Each time the `annotate()` function is executed, it invokes the locally available annotation function Ⓐ. This function then generates a semi-hosting request to the hardware models on a pre-defined I/O port Ⓑ, as illustrated in Listing 5.7. The I/O request transfers the address of

annotation database to the semi-hosting interface, and this address is in fact an offset into the virtual memory allocated to the SystemC memory model. Once the effective address has been calculated, the semi-hosting interface directly obtains the annotation information from the database Ⓒ. The SystemC `wait()` function is invoked to consume the number of processor cycles, as defined in the annotation. The `wait()` call takes place within the processor model context, it updates its local time and gives SystemC an opportunity to re-schedule another hardware process, if necessary. The very same mechanism can be used for any annotation based performance estimation technique.

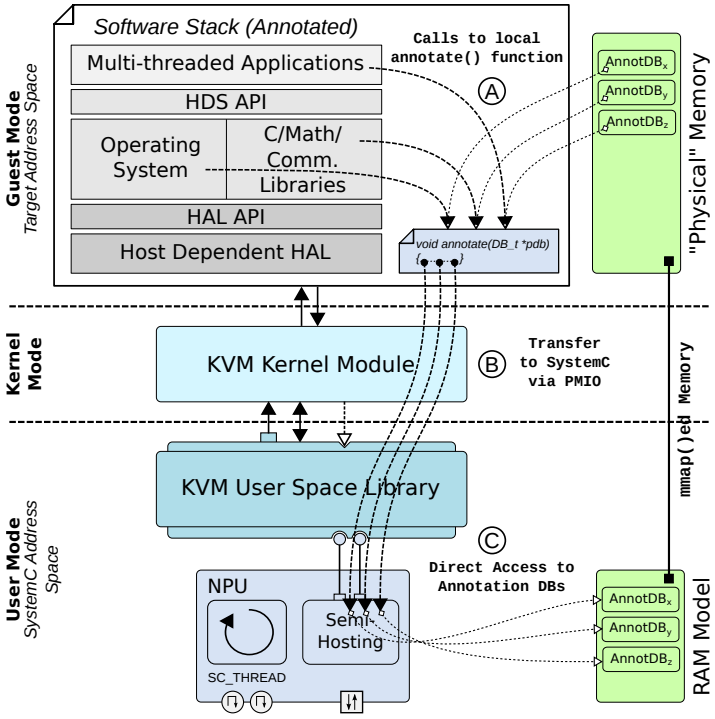


Figure 5.12: Forwarding Annotation Calls to SystemC using PMIO

The annotation technique defines three types of basic blocks: `ENTRY`, `DEFAULT` and `RETURN`. When analyzing the cost of a given software function, these types are used to find out its start and end. In some cases a given target-independent basic block could be split into multiple basic blocks when compiled using the host backend, which makes the annotation call placement a pertinent criterion. Thus, `annotate()` calls are placed at the beginning of `ENTRY` and `DEFAULT` type basic blocks, whereas `annotate()` calls are placed immediately before the return instruction in each `RETURN` type basic block. This change copes with cases when a single `RETURN` type basic block is split into multiple basic blocks during optimization.

As each annotation call requires a guest-to-host mode switch, which is a costly operation and should be avoided, whenever possible. Section 5.3.1 discusses how we can minimize such transitions and improve simulation performance. Chapter 7 will provide results and highlight the cost of such transitions.

Key Point 5.5 Automatic Software Annotation Support

Using the PMIO mechanism supported on HAV-enabled host machines, we can easily integrate any type of software performance estimation technique. For avoiding excessive guest/host mode switches, an annotation buffering scheme on the guest side is highly recommended.

5.3.1 Minimizing Guest *vs.* Host Mode Switches

In HAV based simulation, the guest-to-host mode switching is a costly operation, even though the basic state switch is performed atomically by the hardware. The cost factor comes from the additional state switch performed in by VMM software and includes general purpose, floating point, debug and *Model Specific Registers* (**MSRs**). This design choice by HAV designers is based on the fact that the state of these registers is better known to VMM software, as a result the hardware implementation of state switch is simplified. In addition, the path between VM Exit and VM Entry contributes significantly to this cost, which includes exit reason analysis, in-kernel device and instruction emulation, and user mode support, if required. Evidently all of these operations are costly and slow-down the simulation performance.

In order to achieve faster simulations this cost has to be reduced; either per guest-to-host switch cost or minimizing the total number for switches for a given task. Following list gives the guest events that cause a mode switch and require either kernel mode or user mode support.

❶ Accesses to un-mapped memory or guest paging structures.
❷ Accesses to host specific device models (user-space or in-kernel).
❸ MMIO accesses to SystemC device models.
❹ Synchronization and annotation function calls.
❺ Profiling and debug exceptions.

If the host machine lacks extended page tables support (Section 5.1.2.2) then KVM module provides guest memory virtualization using shadow pages. Maintaining these pages, in correspondence with guest memory accesses, is a costly operation requiring guest mode exiting (❶). On recent machines, EPT support is readily available and native simulation should use EPT enabled KVM kernel module to improve simulation performance.

Host specific device models (❷) are required to make the simulation work on native machines. For example, on x86 machines, device models for LAPIC, PIC, or IOAPIC are required for the guest software to believe that it has access to a real machine. These device models can be either implemented in user-space *i.e.* within the KVM library (outside the framework of SystemC) or provided inside the Linux kernel to avoid user-space accesses. Realism of these device models depends on the guest operating system and its expectations. We will provide more details on this topic in the following sections (*c.f.* Section 5.4.2).

Each MMIO operation may requires access to user-space SystemC models (❸), thus reducing MMIO-induced guest-to-host switches is a desired property. KVM provides a different type of MMIO operation known as *Coalesced* [4] MMIO that can be used to decrease MMIO specific exits. This feature can be particularly useful if the MMIO is uni-directional *i.e.* when the guest software performs write-only operations to certain devices such as the Framebuffer and Terminal devices. Additionally, if writing to such devices does not cause

[4] Unified, united or merged MMIO

any side-effects *i.e.* triggering of some hardware action, then it means that the devices mimic a RAM-like behavior; that is, the MMIO operations can be coalesced. KVM provides the KVM_REGISTER_COALESCED_MMIO API for this purpose, which registers a coalesced MMIO region and is used to combine multiple MMIO requests that are treated on the next exit to user-space.

Synchronizations (❹) and annotations also cause exits to user mode and are mutually exclusive *i.e.* in case of annotations, HAL API based synchronization calls should not be used. Annotations are buffered in guest mode, using a mechanism similar to the coalesced MMIO. This buffering scheme keeps track of basic blocks that are executed by the current software thread. Once a pre-defined threshold is reached, the annotation buffer address is sent to an annotation port, and the annotation mechanism switches to the next available buffer. This scheme helps in minimizing the number of guest-to-host mode switches at the cost of simulation accuracy in multiprocessor platforms. Chapter 7 will provide detailed study of annotation buffering and its influence on simulation performance and accuracy.

Lastly, the profiling and debug exceptions (❺) also require user-mode support, considered minor issues, as they are only used during the debug and profile stages of embedded software development. When we talk about profiling, we refer to host-specific time measurements, for comparing the simulation performance of our native platform *w.r.t.* other native and non-native solutions.

5.4 MPSoC Simulation using Hardware-Assisted Virtualization

As discussed in Chapter 2 that integration of multiple heterogeneous processors into a single SoC is a clear trend in embedded systems. Following this trend, we use KVM to separate the target address space from the user-space, as discussed in Section 5.2.4. The proposed solution has the flexibility to simulate multiple processors in a single SystemC platform. A wide range of multiprocessor architectures can be modeled, such as the tightly coupled (Shared Memory MP, SMP), loosely coupled (Distributed Memory MP) and even the ones in between that share some memory regions only.

The KVM_CREATE_VM API is used to create a virtual machine without any *Virtual CPUs* (**VCPUs**) and memories. Based on the requirement, we may add multiple VCPUs into the virtual machine and share the same memory. If the architecture is fully distributed, we can create multiple virtual machines and connect them together using SystemC interconnection components. KVM also provides the ability to monitor the multiprocessor state of system, enable/disable exceptions and interrupts, and to designate a boot processor. To model a realistic MPSoC platform, support for asynchronous external events, such as interrupts, should be supported as well. We start our discussion on external events and how we can handle them using the HAV approach.

5.4.1 Asynchronous External Events

Asynchronous external events are used to decouple the processor from I/O intensive tasks that can be performed independently, such as a DMA transfer or a block I/O request. On completion of such requests the I/O device signals the processor, and the processor is required to take immediate action by executing the appropriate event handling function.

Without using timing annotations, the native software can only synchronize with the hardware platform when an I/O access takes place, a HAL API call is invoked or an external

event occurs. This corresponds to the loosely-time coding style of TLM 2.0 standard for high simulation speed. Timings annotations and interrupts can help in improving the accuracy of the system, at the cost of more synchronizations, thus reducing simulation performance.

KVM provides functions to set interrupt sensitivity before entering the guest mode, so that the native processor can avoid deadlocks. External events are generated by external device models and are handled by using the event injection mechanism supported by KVM. Such events are delivered immediately after the next entry to guest mode and the VCPU takes appropriate action. Figure 5.13 details the steps used for delivering such an external event from SystemC models to the actual interrupt handler in guest mode:

① An external event is generated by a SystemC model, such as a `TIMER` and is delivered to the interrupt control logic (`ITC`) inside the native processor.

② The interrupt controller decides, considering the interrupt configurations (enable/disable flags, masks *etc.*), whether to deliver the interrupt or mask it. If the interrupt is to be delivered, it sends an interrupt injection request to the KVM library.

③ KVM library requests the KVM module for interrupt injection using the `KVM_IRQ_LINE ioctl()` and provides the `IRQ` line number to be asserted/de-asserted. The same `KVM_IRQ_LINE ioctl()` can be used to check for the interrupt delivery status of previously requested external events.

④ KVM module maintains models of PIC and IOAPIC (on x86) and setups the interrupt pending request on both of them. Interrupt delivery to software depends on whether the guest software uses PIC or IOAPIC for interrupt management. The actual interrupt injection takes place on the next VM Entry, if the guest is ready to receive interrupts *i.e.* `EFLAGS.IF = 1`, otherwise the VMM can request interrupt-window exiting using the virtual machine execution control in VMCS. The guest will perform a VM Exit as soon as it is ready to receive external events.

⑤ The VCPU receives the interrupt immediately after entry to guest mode and invokes the appropriate interrupt handler using the guest *Interrupt Descriptor Table* (**IDT**), just as if the interrupt had actually occurred immediately after the VM Entry.

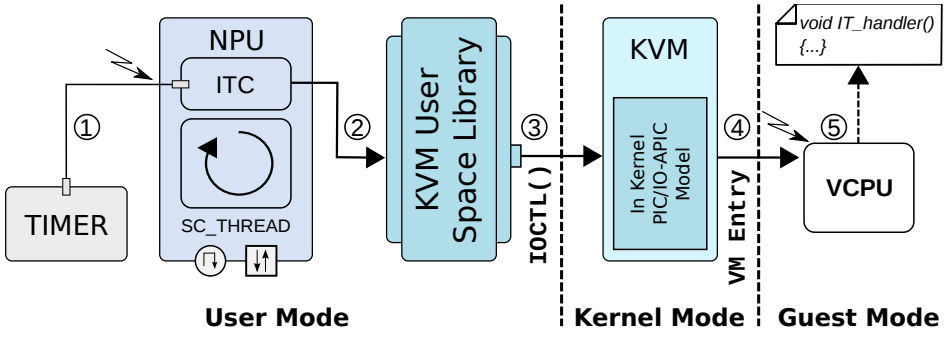


Figure 5.13: Handling External Events using Interrupt Injection in KVM

5.4.2 Simulating Multiple Processors

Simulating multiple processors is a basic requirement for architecture exploration in MPSoC context. We address this problem by using the un-modified version of KVM and implement the required infrastructure around it to obtain a functional system, from the guest software point of view only. This model is very similar to virtual machine based solution and uses the SystemC components as I/O devices. In addition to the SystemC thread, every processor creates a POSIX thread to avoid modifications to the KVM module, as it expects host threads instead of SystemC threads. Details on how we remove the host threads to obtain a true native MPSoC simulation platform, will be discussed later (*c.f*. Section 5.4.3).

5.4.2.1 Host Specific Processor Initialization

A key design feature of a VMM is to hide its existence from guest software view and mimic the real machine behavior. The guest software thus needs to follow the real machine semantics when initializing system resources, including processors and I/O devices. In mono-processor context, this process is simple and has to deal with few details of host machine specific initializations such as real and protected mode semantics of the boot process, on x86 hosts.

When implementing an SMP simulation solution, the guest software has to follow the boot protocol of the host machine; to be more specific the MP Initialization protocol on x86 machines (Chapter#8 in [Int11]). This protocol specifies a sequence of interrupts that must be sent by the bootstrap processor to other processors in a given real or virtual machine. These interrupts include the Init and Startup *Inter-Processor Interrupts* (**IPIs**) to modify the destination processor's state.

Initially, every VCPU starts in the `UNINITIALIZED` state, except the boot VCPU, which is `RUNNABLE` by default. All other VCPUs wait for interrupts and proceed to `INIT_RECEIVED`, `SIPI_RECEIVED` and `RUNNABLE` states upon receiving the corresponding signals from boot VCPU. This functionality is provided by the *Local APIC* (**LAPIC**) device model, as part of the APIC virtualization within the KVM kernel module. The boot code programs the LAPIC model and once all non-boot processors VCPUs have been initialized, the boot VCPU proceeds to the operating system initialization. Non-boot VCPUs wait for the `CPU_MP_PROCEED` HAL API call from the boot VCPU, before executing any software thread.

5.4.2.2 SMP Compatible Hardware Abstraction Layer

The HAL layer is the key component that hides the hardware specific details and directly interacts with the VMM. In order to simulate a multiprocessor machine, the HAL has to provide APIs that could provide each VCPU with it identification and ability to synchronize with other processors. KVM provides an in-kernel *Local APIC* (**LAPIC**) device model for each VCPU instance. In multiprocessor context the HAL layer has to interact with this device model using the MMIO interface for implementing the SMP specific functions. Listing 5.8 provides an example of how we can get the current VCPU `ID`, using the `CPU_MP_ID` HAL API call from the guest software. The HAL provides another key API *i.e*. `CPU_MP_SEND_IPI` for sending IPIs to VCPUs, including itself. This API provides the guest operating system with the ability to suspend and dispatch software threads onto the available processors. The same HAL API is used for enabling/disabling interrupts on different VCPUs.

In order to be fully compatible with the SystemC threads, none of the software functions should use a spinlock that tries to take a global lock held on another processor. This situation

Listing 5.8 Getting the VCPU ID on an x86 Machine using Local APIC MMIO

```
1       .globl CPU_MP_ID
2   CPU_MP_ID:
3       movl    0xFEE00020, %eax        /* Read the Local APIC ID Register */
4       shrl    $0x18, %eax             /* Drop the 24 LSBs by Right Shift */
5       ret                             /* Return EAX with the 8 MSBs i.e. VCPU-ID */
```

does not arise in the case of POSIX threads, as KVM can re-schedule another thread if a thread gets blocked on some memory-mapped resource.

In our HAV based solution, the HAL layer is implemented for DNA-OS, and this situation arises in `CPU_MP_WAIT` and `CPU_TEST_AND_SET` HAL API calls. In the first case, we know that the `CPU_MP_WAIT` API has been executed by a non-boot processor, thus we can safely suspend the execution of current VCPU and `SC_THREAD` until it receives the Init and Startup IPIs. We implement this feature by sending an I/O request to a designated PMIO port and ask SystemC to put the current native processor to sleep and wake another one. In the second case, for `CPU_TEST_AND_SET` API, the guest software thread currently holding the lock is unknown. However, we can find the processor ID on which the lock-holder thread is running, by saving the processor ID in the lock. Once the lock holder processor is known, we can suspend the execution of lock requester processor in SystemC and let others continue their execution until the lock is released and SystemC re-schedules the lock requester processor.

In order to make sure that SystemC does not re-schedule the lock requester processor immediately, the simulated time of the lock requester processor is advanced to match the lock holder processor's next wake-up time. Figure 5.14 explains this problem, where a guest software thread in group "C" is currently holding the global spinlock and a thread in group "A" on the currently active VCPU is trying to acquire the same lock using the `CPU_TEST_AND_SET` instruction. This situation will lead to a deadlock on the current VCPU and simulation will freeze unless the software is annotated, or the control is explicitly returned to the SystemC scheduler.

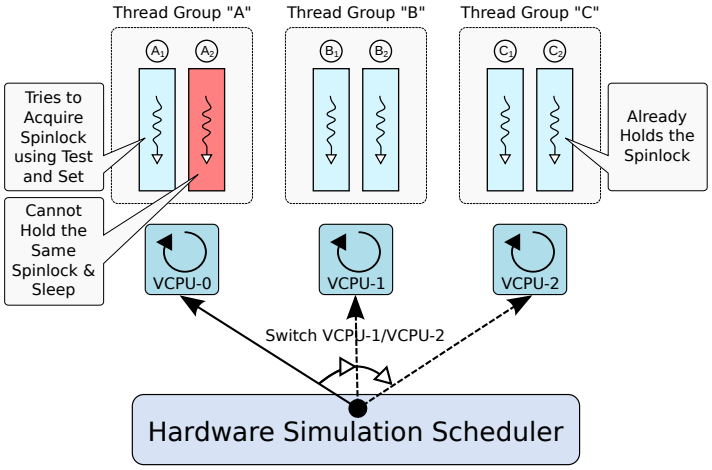


Figure 5.14: Guest Software Threads and VCPU Locking Issue

To avoid the deadlock situation, no guest software thread should hold a global spinlock on a VCPU and sleep, as other software threads may try to acquire the same lock. For example, in Figure 5.14 the thread A_2 cannot hold the same lock while the thread A_1 is trying to

acquire it. As a rule of thumb the lock holder thread should not sleep while holding the lock, thus avoiding the possibility of getting scheduled-out and potentially deadlocking other threads on the same processor. Section 5.4.3 will propose a different implementation of the `CPU_TEST_AND_SET` HAL API for use with SystemC threads only for obtaining correct timing behavior.

Using the SMP compatible HAL, along-with un-modified KVM kernel module and POSIX threads, the simulation platform can provide functional execution support to the guest software and uses SystemC device models for performing I/O operations. A key deficiency in this model, comes from the sequential execution of VCPUs and simulated processor timings in SystemC. Each processor contributes to the global simulated time instead of its own local time, as the VCPU scheduling is under KVM control. Nevertheless this model is sufficient to show that multiprocessor modeling is feasible and the HAL implementation is correct for execution within the virtual machine environment. Section 5.4.3 addresses the POSIX threads issue and the necessary changes to KVM, HAL and SystemC processor models to provide a true MPSoC native simulation platform.

5.4.2.3 Hardware-Assisted Software Debugging

To make the simulation platform usable from software development point of view, debugging support is an essential requirement for both operating system and application level codes.

KVM provides the ability to get and set general purpose, system and debug registers for each VCPU in the virtual machine, using the `KVM_GET_[REGS|SREGS|DEBUGREGS]` and `KVM_SET_[REGS|SREGS|DEBUGREGS] ioctl()s`. Similarly, the `KVM_GUESTDBG_USE_HW_BP` and `KVM_GUESTDBG_USE_SW_BP` debug control flags can be used to activate the hardware [5] and software [6] breakpoints in guest mode. These features provide the bare-bones necessary for providing debug support, but KVM user mode library lacks a debugging tool to efficiently exploit these capabilities.

A GDB server implementation that can communicate with standard host debugger using sockets, has been introduced in the proposed simulation framework. Once the debugging mode is enabled using the `KVM_GUESTDBG_ENABLE` debug control flag, simulation control is forwarded to the debugging interface on each debug exception in guest mode, as shown in Figure 5.4 Ⓗ. Once this capability is available, the guest software can be debugged from the first boot instruction and all necessary support for setting breakpoints, examining memory contents, modifications to the VCPU registers is included in the GDB server.

In order to debug complex hardware software interactions one cannot rely on software only debuggers. In such situations, the GDB server can be used (for software debugging) in combination with the standard host debugger (for hardware debugging) to understand the source of difficult problems. One key caveat still remains a problematic case and is related to the debugging of KVM module. It is usually difficult due to the kernel mode restrictions, and one has to rely on kernel logs or setup multiple machines for this purpose.

[5] A hardware breakpoint uses dedicated debug registers and does not modify the memory contents. Hardware breakpoints are architecture specific and limited in number *e.g.* x86 allows upto 4 hardware breakpoints.

[6] A software breakpoint inserts an invalid instruction in memory and then handles the exception, replaces the original memory contents and then singles steps over the instruction.

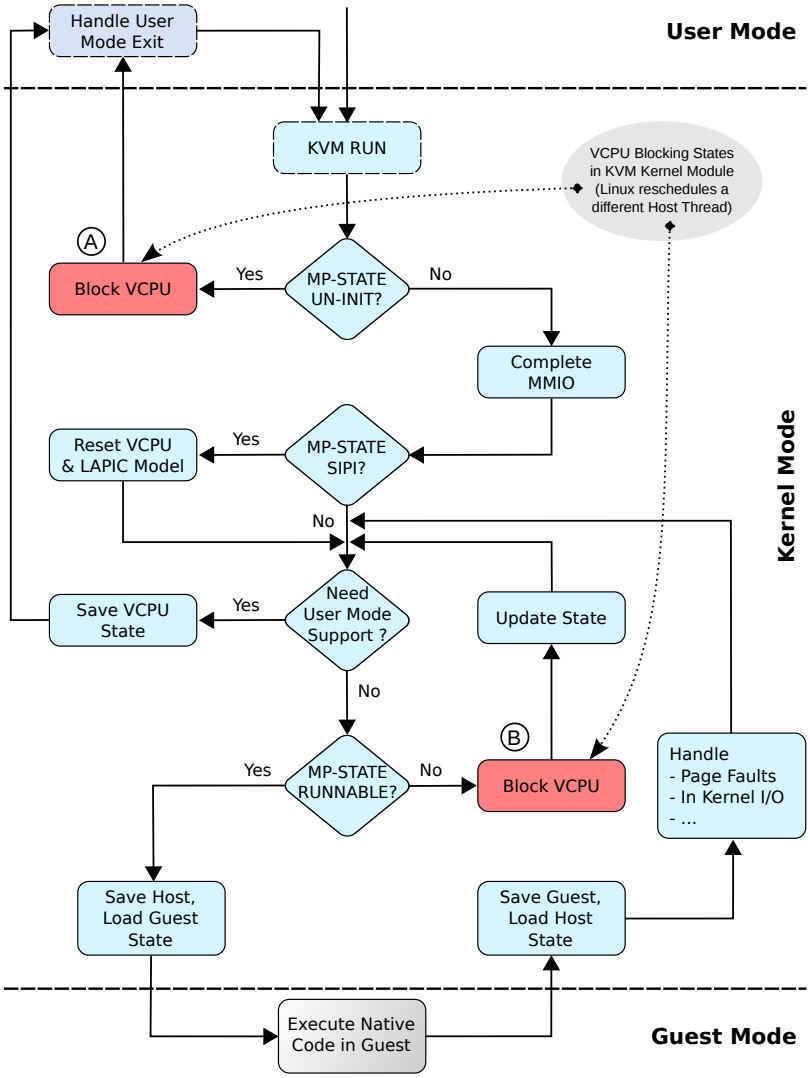


Figure 5.15: VCPU Execution Flow within KVM and Blocking States

5.4.3 Virtual CPU Scheduling in Kernel Virtual Machine

KVM expects each VCPU to execute within its own thread on the host machine, therefore it can schedule each VCPU under the guest software control and independently from one another. For example when a non-boot VCPU starts execution, it is in the `UNINITIALIZED` state, as discussed in Section 5.4.2.1, as opposed to the boot VCPU which is `RUNNABLE`. KVM cannot allow non-boot VCPUs to continue their execution while in `UNINITIALIZED` state, so it blocks them on a wait queue until the boot VCPU sends an Init and Startup IPI. Moreover, when a guest processor has no work to do, it switches to the idle thread and executes the `HLT` instruction. KVM blocks such VCPUs and calls the host system scheduler to resume another runnable process, a VCPU or another host system thread. Figure 5.15 illustrates the VCPU execution flow within KVM and highlights the blocking states where VCPU threads can be scheduled out and put in wait queues.

VCPU blocking is a problem from the simulation perspective as a blocked VCPU halts the

whole simulator process, and the simulation cannot continue with other hardware threads. This limitation comes from the fact that SystemC simulation runs within a single host thread, although many SC_THREADs are present within the simulator. To solve this problem, we remove these blocking states and give control back to SystemC processor model with special error codes. The blocking state Ⓐ in Figure 5.15 is simple to handle, as KVM can return to the SystemC processor model with VCPU_BLOCK_SELF error code. The blocking state Ⓑ can result from either the VCPU entering the idle thread or when the current VCPU sends an IPI to a different processor. In case of idle state, KVM returns with the same VCPU_BLOCK_SELF error code and the SystemC processor model starts waiting on the RUNNABLE_EVENT. The RUNNABLE_EVENT event is triggered when a processor sends an IPI to the current processor. In case the current VCPU sends an IPI to another VCPU, the KVM arrives at state Ⓑ as well. The modified KVM gives control back to the SystemC processor model with target processor ID and the RUNNABLE_EVENT of target processor is triggered. In order to ensure that the target processor gets re-scheduled and enters the guest mode before the current processor a SC_ZERO_TIME wait is introduced for the current processor. This wait ensures that the current processor gets re-scheduled *after* the target processor has entered the guest mode at-least once. Simulation can halt if the current processor sends two consecutive IPIs to the same target processor, and the target processor does not get a chance to execute the IPI handler and release the IPI lock, producing a deadlock.

Section 5.4.2.2 discussed the need to adapt HAL APIs when using SystemC threads only. Two APIs were identified *i.e.* CPU_MP_WAIT and CPU_TEST_AND_SET for DNA-OS. The first API behaves much like the VCPU_BLOCK_SELF error condition introduced in Figure 5.15 Ⓐ except that the guest software gets blocked on a global lock in this case. From KVM point of view, the non-boot VCPU has been properly initialized and is now in RUNNABLE state and its the HAL API that has to inform SystemC about the blocking condition.

The problem introduced by the second HAL API was illustrated in Figure 5.14, and we propose a modified implementation of CPU_TEST_AND_SET API in Listing 5.9. Key idea is to try to store the current VCPU ID+1 for taking the global lock and on failure, inform the SystemC processor model about the situation. The lock requester VCPU's processor model receives the ID of VCPU currently holding the lock, and advances its own simulated time to the next wake-up time of lock holding processor. Figure 5.16 shows an example where VCPU-0 takes a locks at time t_1 and VCPU-1 tries to acquire the same lock at time t_3. In order to know the next wake-up time of a given processor, each processor model is required to update this information before calling the SystemC wait() function. The lock requester VCPU-1 at time t_3 does not know when the lock will be released by VCPU-0, but it can nevertheless advance its simulated time to the next wake-up time, t_5 in this example. This solution avoids deadlocks as SystemC scheduler will schedule VCPU-2 instead of VCPU-1 and simulation will continue.

In situations when the lock requester VCPU is re-scheduled and the lock holder VCPU is still holding the global lock, the native processor will use a SC_ZERO_TIME wait to invoke the SystemC scheduler and cause scheduling of another VCPU. In certain cases, the SystemC scheduler can re-schedule the lock holder VCPU *before* its expected wake-up time, for example when it receives an IPI, and this situation would cause in-accuracies in the simulated time of lock requester VCPU.

Once the above-mentioned changes are incorporated in processor models, KVM kernel module and HAL APIs, simulation of a complete MPSoC platform is possible. Using this model, simulated time is properly managed under SystemC control as only SC_THREADs are used.

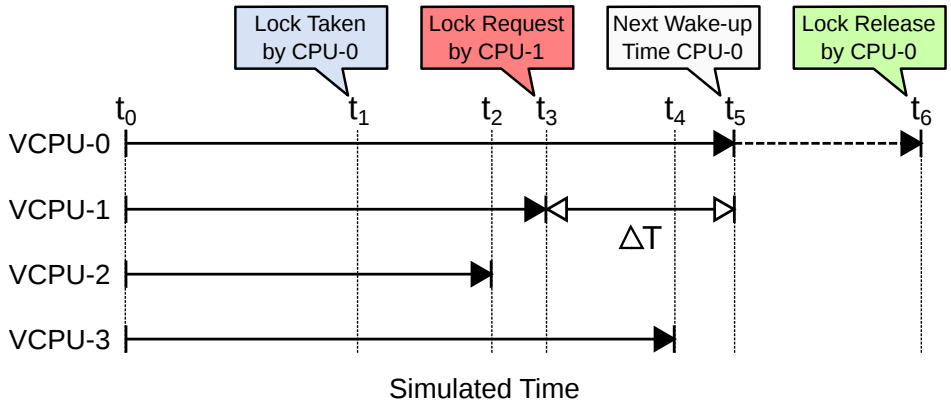


Figure 5.16: SystemC Timing Modifications on Execution of Test and Set (Unsuccessful Case)

Creation of multiple virtual machines with different number of processors is possible and is shown as *Processor Wrappers* (**PWs**) in Figure 5.17. Chapter 7 will provide experimental evidence and validate the proposed solution.

Key Point 5.6 Inter-processor Locking and Memory Mapped Resources

Software threads cannot wait on locks held by other processors, as only one processor is simulated at any instant. As a general rule, software should not block on any memory mapped resource without advancing its own simulated time, thus giving hardware simulation kernel an opportunity to re-schedule another processor and avoid deadlocks.

Listing 5.9 HAL Function `CPU_TEST_AND_SET()` for Native Simulation (SystemC threads)

```
1  #define SYSTEMC_TEST_N_SET_PORT 0x3000
2
3  long int CPU_TEST_AND_SET (volatile long int * spinlock)
4  {
5      long int oldval = 0;                    /* Previous value to compare with. */
6      int my_cpu_id = CPU_MP_ID() + 1;        /* VCPU-ID + 1 for taking the spinlock. */
7      int locker_cpu_id = 0;                  /* Zero means unlocked */
8
9      __asm__ volatile (
10         "lock cmpxchgl %1, %2\n"
11             : "=a" (locker_cpu_id)      /* Non-zero means locked by another VCPU */
12             : "r" (my_cpu_id), "m" (*spinlock), "0" (oldval)
13             : "memory");
14
15     if(locker_cpu_id != 0 && locker_cpu_id != my_cpu_id)
16     {
17         /* The lock is taken by someone else; tell SystemC about it */
18         CPU_IO_WRITE(UINT32, SYSTEMC_TEST_N_SET_PORT, locker_cpu_id);
19         return 1;    /* Try again after the other VCPUs have run for some time */
20     }
21
22     return 0;        /* OK, the current VCPU has this lock */
23 }
```

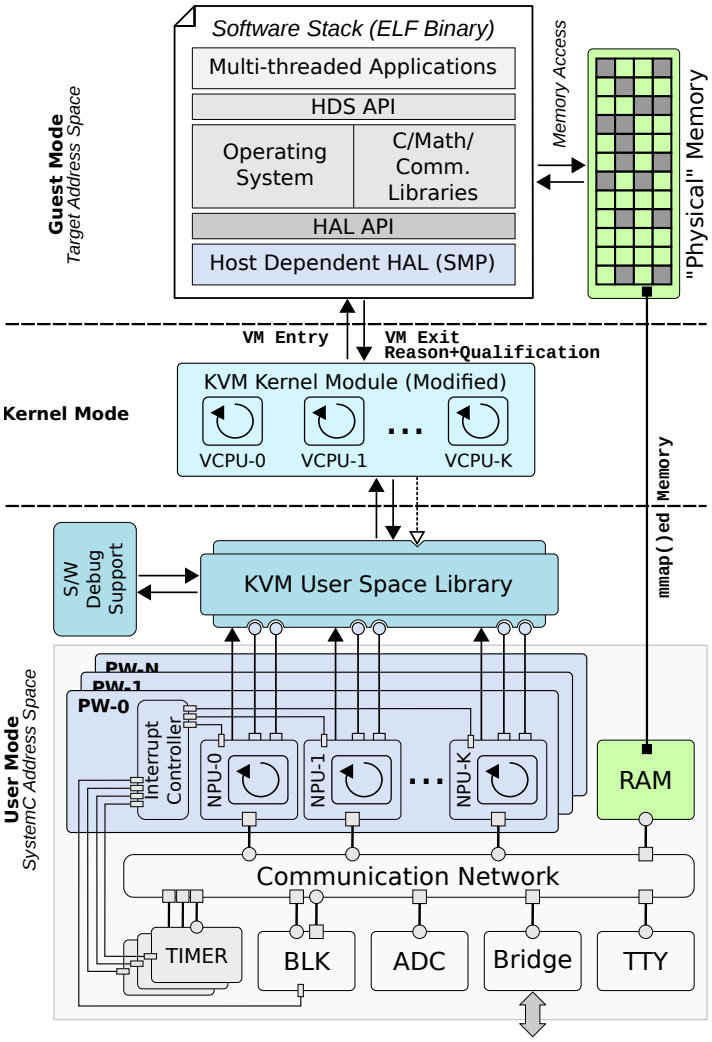


Figure 5.17: Multiprocessor Simulation using KVM with Debug and Interrupt Support

5.5 Hybrid MPSoC Simulation

In embedded systems domain, heterogeneous multi-core architectures have been used for years to reach the required level of performance for a given power budget. Simulation platforms that can model such architectures are difficult to develop, as different simulation technologies may be used to model different types of processing elements. For example, an ARM-based GPP along-with a DSP on top of the same simulation platform. Different types of simulation technologies can be used for hybrid simulations, such as DBT [Bel05, GFP09] for GPP and native simulation techniques [GHP09, PV09] for DSP processors.

Primary difficulty in such models comes from the memory and I/O address-space representations in software and hardware models. For instance, DBT uses cross-compiled software with target addresses for memory and I/O operations and expects the platform models to simulate the same addresses. Traditional native techniques use the host system addresses for all memory and I/O accesses, either by using host operating system exception mechanisms or using the modified hardware platform models.

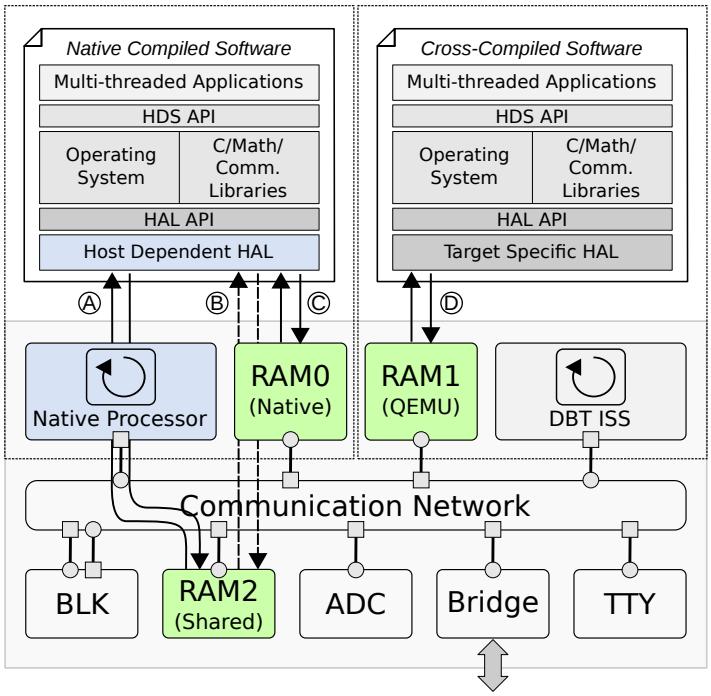


Figure 5.18: Hybrid Simulation Platform using DBT and Native Processors

The HAV-based native simulation approach can solve this problem, as the software stack uses target addresses and does not require modifications to the platform device models. Thus, the same set of device models are shared between DBT based ISSes and native processors. In contrast to the proposed KVM solution, traditional native simulation solutions are not transparent to the embedded OS and applications. The simulated software needs to know the host addresses used by the simulation platform. This aspect makes the simulated software difficult to design and very difficult to integrate with other target address based simulators, such as QEMU.

The hybrid MPSoC simulation platforms with different kinds of processing elements have as advantage the ability to put together existing processing elements using different simulation technologies. Already available and tested processor models can be re-used in hybrid simulation, and a better match between processor type and available simulation technology results in increased performance and accuracy of the simulated design. Figure 5.18 shows an example hybrid platform that uses two different types of processing elements along-with their local memories *i.e.* `RAM0` and `RAM1` and corresponding access methods. KVM uses memory mapping Ⓒ and QEMU uses back-door access mechanism for accessing local memories Ⓓ.

The communication mechanism between these processors is also important, in order to build a useful simulation platform. Here, the I/O accesses use the original mechanisms supported by the processors, as the shared platform components that provide the same interfaces to both of the processors. However, shared memory accesses, such as in `RAM2` device model in Figure 5.18, can have two possible implementations for the native processors.

- ❖ MMIO Address Space: The shared memory component can be accessed from the native processor using the MMIO address space along-with normal I/O accesses, as shown

in Figure 5.18 Ⓐ. This choice is costly in terms of simulation performance, as each memory access requires a guest-to-host mode switch, but can be more accurate as each shared memory access renders control to the SystemC kernel with a re-scheduling opportunity.

- HAV-based Memory Mapping: The native processor can also establish a direct memory mapping to the shared memory model using the `KVM_SET_USER_MEMORY_REGION` capability as shown in Listing 5.3 and Figure 5.18 Ⓑ. This choice is better in terms of simulation performance, but can increase simulation inaccuracy, as the software execution remains in guest mode for all shared memory accesses.

Both choices are transparent from the embedded OS and application point of views. From the embedded software point of view, QEMU and KVM are identical and use the same target address space, which makes it possible to partition and map one application onto two different processors and perform communication using shared memories. To the best of our knowledge, there exists no other native solution that can provide such high degree of flexibility, by using the HAV-based transparent address translation and supports this kind-of hybrid simulation models.

Endianness is an issue for the native processors in shared memory based communication contexts. As the native binary is based on the host instruction set, the endianness is fixed to either little or big endian. It is possible for the endianness of MPSoC platform to be different from that of the host machine. The data and private memory accesses will have the same endianness as the host processor. For I/O accesses, the native processor would be responsible for swapping the endianness if the MPSoC platform has different endianness as compared to the host processor. Due to the use of *Direct Memory Interface* (**DMI**), sharing memory between a native simulator and an instruction accurate ISS of different endianness is still an open problem.

5.5.1 Memory and I/O Access Comparison

This section briefly compares the memory and I/O address space access mechanisms used in different simulation technologies. We compare the DBT-based QEMU, traditional Native techniques and our proposed KVM-based native simulation platform. Three types of accesses are compared including Local Memories, I/O Accesses and Shared Memories within these techniques. Table 5.2 provides examples, addressing details and access implementations for each of these technologies.

5.6 Conclusions and Limitations

This chapter presented the main contribution of this thesis. The first section was dedicated to the essential concepts of *Hardware-Assisted Virtualization* (**HAV**) that provide the basis for our solution. We demonstrated that the address spaces problem introduced by the native techniques can be solved efficiently using the HAV technology. Key elements of the proposed solution including, *Native Processing Units* (**NPUs**), Host Dependent HAL and I/O accesses using *Memory-Mapped I/O* (**MMIO**) capability of KVM were discussed, in mono- and multi-processor contexts.

Key benefits of the proposed solution include:

- Transparent memory accesses in software and mapping of exactly the same memory address space to hardware memory models allowing for optimal bi-directional memory accesses from software, as well as hardware models.
- Validation of a significant amount of software stack, including operating system and applications, for the target architecture except the HAL implementation.
- Absence of software coding constraints such as the use of hard-coded addresses, or specialized linker scripts.
- Use of un-modified hardware IP models and statically compiled software stack requiring no dynamic linking or patching support.
- Native simulation of MMU-based operating systems is possible, as KVM provides the required address translation in guest mode using SPT or EPT support.
- Support for automatic instrumentation techniques using PMIO or similar mechanisms on host machines.
- Simulation of multiple processing elements and flexibility to model hybrid solutions on top of a shared MPSoC platform.

And key limitations of the proposed solution are:

- All hardware/software interactions must take place through the HAL layer except for memory accesses. This limitation requires a guest operating system specific implementation of the HAL API, which could be a problem in case of complex operating systems such as Linux.
- Software execution remains in guest mode for all compute and memory intensive tasks, but I/O accesses require a guest-to-host mode switch, which is a costly operation and degrades simulation performance.
- Support for *Self-Modifying Code* (**SMC**) is not available in our proposed solution, as the software stack is compiled statically and no run-time translation support is available.
- Simulation accuracy from performance estimation perspective is highly-dependent on annotation accuracy; thus performance estimation of complex architectures is difficult using traditional annotation techniques.

Although our proposed solution has certain limitations but the interest of this approach and benefits outweigh these limitations. Next chapter will present a *Static Binary Translation* (**SBT**) technique for generating native code and simulating complex VLIW architectures on top of the simulation platform proposed in this chapter.

Local Memory Read/Write Access			
	Code Example	***Address of variable i***	***Access Implementation***
QEMU	int i = 55; &i=?	Target memory address	Address translated during the basic block translation.
Native	int i = 55; &i=?	Host SystemC memory module address	Accessed directly without translation.
KVM	int i = 55; &i=?	Target memory address	Accessed directly using MMU mapping of the host processor.
Shared Memory Read/Write Access			
	Code Example	***Address of variable i***	***Access Implementation***
QEMU	*(int *)0xBF001018 = 0x66;	Target memory or I/O address	Same as the Memory or I/O access.
Native	addr = SHARED_MEM_BASE + offset; *(int *)addr = 0x66;	Host SystemC shared memory module address	Same as the Memory access.
KVM	*(int *)0xBF001018 = 0x66;	Target memory or I/O address	Same as the Memory or I/O access.
I/O Read/Write Access			
	Code Example	***Address of variable i***	***Access Implementation***
QEMU	*(UART_ADDR) = 0x55;	Target I/O address	Access to the SystemC UART module by QEMU.
Native	CPU_IO_WRITE(UART_ADDR, 0x55);	Host SystemC UART module address	Access to the SystemC UART module using special functions.
KVM	*(UART_ADDR) = 0x55;	Target I/O address	Access to the SystemC UART module by KVM.

Table 5.2: Local Memories, I/O and Shared Memory Accesses in QEMU, Traditional Native and KVM-based Platforms

Everything should be made as simple as possible, but not simpler.

Albert Einstein

6 Static Binary Translation Targeting VLIW Processor Simulation

Simulation of VLIW processors on scalar machines is an interesting topic, as it involves many architectural aspects rarely found in RISC machines. This chapter presents a *Static Binary Translation* (**SBT**) flow for VLIW processor simulation on x86 based native platforms. The translation framework generates native code from cross-compiled VLIW executables and simulates them on the HAV based platform presented in the previous chapter. This approach is interesting in situations where either the source code is not available or the target platform is not supported by any retargetable compilation framework, which is usually the case for VLIW processors. Specialized toolchains are used to generate code for VLIW processors, applying many target specific optimizations in the process. The SBT approach has been implemented for *Texas Instruments* (**TI**) C6x series processors, as these processors present some of the most interesting features found in VLIW architectures.

6.1 Static Binary Translation Principle and Constraints

Dynamic translators perform all of the processor pipeline stages at runtime and frequently use a translation cache to amortize the cost of instruction fetch and decoding steps [SL98, ZT00, Bel05]. Static translators separate the instruction fetch and decode stages from execution, thus improving the runtime performance of generated simulators [CVE00, CYH+08, SCHY12].

Basic static translation principle is highlighted in Figure 4.7, where the translation steps are decoupled into two distinct phases. Firstly, the input instructions are converted to an *Intermediate Representation* (**IR**) that is independent of both *source* and *target* machine architectures, using a translation front-end. Secondly, the IR is converted to the target machine-specific representation using a code generation backend. Using this principle, a significant improvement in simulation performance can be achieved, as the expensive decoding and translation stages are performed only once. At runtime, the simulator executes

the translated instructions as many times as necessary, without any translation overheads.

We extend the same principle and propose a static translation flow for VLIW processors. We use the *Low Level Virtual Machine* (**LLVM**)-IR as an intermediate language for the translation process. This choice provides us with three key benefits. Firstly, the translation flow becomes retargetable as LLVM-IR is independent of both source and target architectures. Secondly, the existing optimizations of LLVM infrastructure can be re-used to optimize the quality of generated code and lastly the LLVM-IR provides a set of infinite virtual registers, which bridge the architectural differences between source and target architectures. For example, the TI C64x processors have 64 general purpose registers and mapping all of these registers directly to an x86 machine is rather difficult, as it has only 8 general purpose registers.

The proposed SBT technique ensures that the generated IR is correct *w.r.t.* to the VLIW architectural semantics, as discussed in Section 3.6. We rely on LLVM optimization passes to remove unnecessary instructions during optimization phase.

6.1.1 Static Translation Specific Constraints

Static translation performs everything at compile time, and it has some limitations *w.r.t.* to dynamic code behaviors. Static branch targets can be easily identified and analyzed during translation, as opposed to indirect branch instructions. There are two principle options for handling indirect branch instructions *i.e.* either provide dynamic translation support or do the static translation of all execute packets in addition to the higher level aggregations *e.g.* VLIW basic blocks.

In certain cases, VLIW binaries can contain hand-written assembly code, with branch instructions targeting the middle of execute packets, for optimization or code-size reasons. Usually, VLIW compilers cannot produce this type of binary code, so we do not handle such cases in the proposed translation flow. *Self-Modifying Code* (**SMC**) is another issue, which usually emerges due to the presence of pointers and dynamic linking in software. Usually, VLIW binaries do not contain such code segments, as most of the functionality is processing intensive. In VLIW binaries, instructions are rarely modified during execution, so we do not handle such cases as well.

6.1.2 Virtualization Specific Constraints

Native simulation requires that simulated software and hardware models share a uniform view of system memory. For example, when software programs a DMA device to copy a software allocated buffer into a hardware device, consistency of addresses must be assured. Our HAV based native simulation platform shown in Figure 5.17, provides the capability to simulate software in target address space and access simulated memory transparently. In this virtualized memory context, runtime support from host operating system is not available and the software executes as if running on a baremetal machine, limiting the possibility of dynamic translations. For example, if the simulated software wishes to invoke a host function, it cannot do it directly and has to rely on alternative means, such as semi-hosting support as discussed in Section 5.2.4.2.

6.2 Retargetable Static Translation of VLIW Software

Our solution to the simulation for VLIW processors, relies on HAV based native platform proposed in Section 5.2.4.3. The native platform provides the capability to access simulated memory in target address space, without requiring any changes to hardware models and software stack. The proposed static translation flow for VLIW processors is shown in Figure 6.1. This design flow is based on a set components from LLVM compiler infrastructure. Thus, it re-uses many of the already available optimization and code generation components. Broadly speaking, the proposed translation flow uses four types of components:

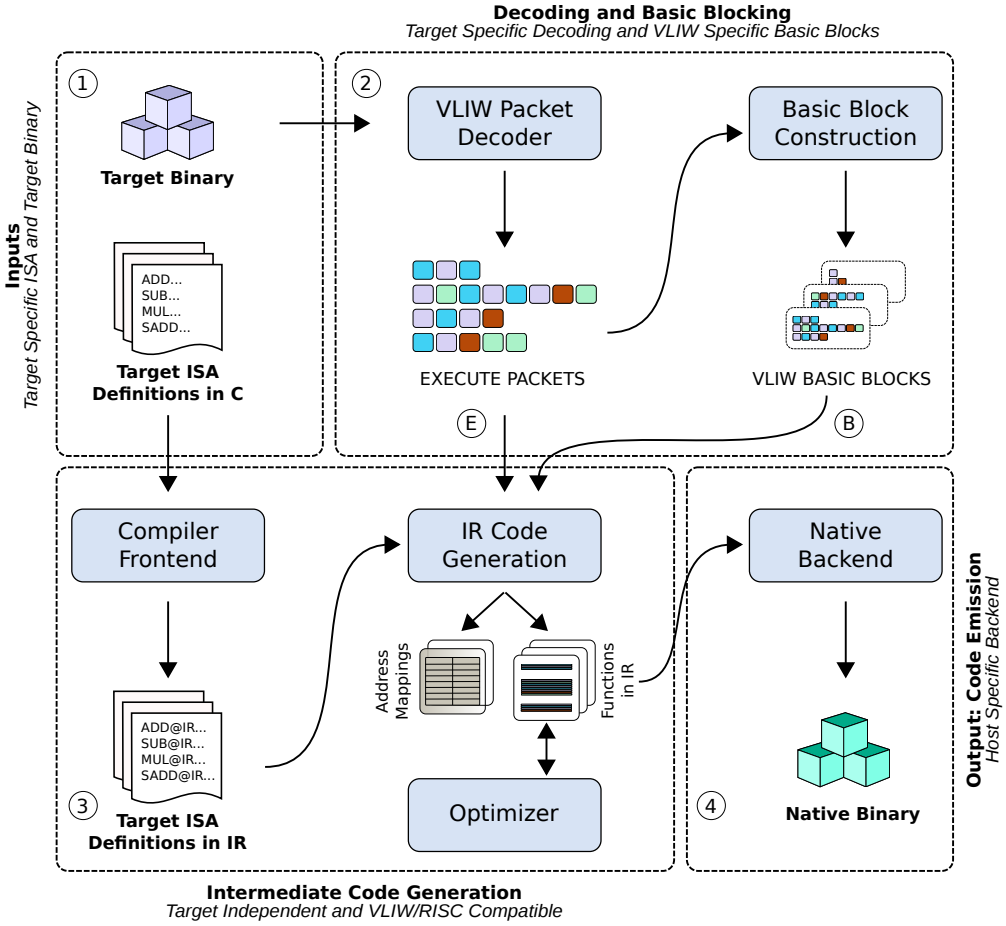


Figure 6.1: Static Binary Translation Flow for VLIW Binaries

① This set contains target binary to be simulated and a collection of high-level functions describing the behavior of target *Instruction Set Architecture* (**ISA**). The target binary is generated from the target-specific toolchain, including all optimizations. Section 6.2.1 gives more detail on ISA behavior definition.

② These include components that are specific to target architecture for instruction decoding and adapted to VLIW architectures. The VLIW packet decoder implements the binary decoding logic as well as extracts parallelism, forming decoded execute packets. Section 6.2.2 will provide more details on this subject.

③ These components include IR code generation modules, which take as input the target ISA behavior in IR and generate intermediate functions from the execute packets and basic blocks. The generated IR code can be optionally optimized using components in this set. Section 6.2.3 gives detailed description of this process.

④ This set includes components for the final code generation. Inputs to this stage are in LLVM-IR, so many different types of backends can be used, making it a retargetable translation flow. We use the x86 backend to generate native code and simulate it on our HAV based simulation platform.

The SBT proposed flow is generic and can be easily adapted for RISC processors. Only three components need to be modified *i.e.* the ISA definitions, the instruction decoder and basic block construction module. Rest of the static translation flow remains unchanged.

6.2.1 Instruction Representation

The VLIW ISA definitions in Figure 6.1 ① are provided in 'C' language and used to generate ISA definitions in LLVM-IR, using the compiler front-end. Each target instruction has a specific behavior that is *how* and *when* it modifies the register, memory or control state of the processor. Once the ISA is available in LLVM-IR, it can be used to compose LLVM-IR code for VLIW basic blocks and execute packets.

Each VLIW instruction is represented with multiple definitions in LLVM-IR, exhaustively covering all operand type combinations. This strategy simplifies the ISA definition process, at the cost of size, which becomes irrelevant once definition generation process is automated. In addition, it makes the optimization step easier and more robust as there are fewer if/else/switch structures to be considered. For example, if an ADD instruction takes two operand type combinations such as: `ADD UInt32, UInt32, UInt32` and `ADD UInt32, UInt40, UInt40`, then two corresponding ISA definitions will be given in 'C' and converted to LLVM-IR *e.g.* `ADD_UR32_UR32_UR32(){...}` and `ADD_UR32_UR40_UR40(){...}`. Table 6.1 gives the operand type naming convention used in ISA definitions. Listing 6.1 gives an example ISA definition in 'C'.

Each ISA definition represents a macro-operation as compared to micro-operations in QEMU [Bel05] *i.e.* an ISA definition specifies the complete behavior of a VLIW instruction for a particular operand type combination, except for the instruction result storage. The instruction result affectation is delegated to the IR code generation steps (*c.f.* Section 6.2.3). For example, Lines 8 and 9 in Listing 6.1 read input operands from processor state, Line 10 specifies the instruction behavior and Line 11 saves the output in the `result` pointer passed by the IR code generator. Line 13 indicates a normal instruction completion.

Operand Type	Register (R)	Constant (C)
Unsigned (U)	_UR<#ofbits>	_UC<#ofbits>
Signed (S)	_SR<#ofbits>	_SC<#ofbits>

Table 6.1: Operand Types in ISA Definitions

Listing 6.1 An Example ISA Definition in C

```
1  /// MPYSU - Multiply Signed 16 LSB x Unsigned 16 LSB.
2  uint32_t C6xMPYSU_SC5_UR16_SR32(C6x_DSPState_t * p_state, uint8_t is_cond,
3      uint8_t be_zero, uint16_t idx_rc, uint32_t constant, uint16_t idx_rb,
4      uint16_t idx_rd, C6x_Result_t * result)
5  {
6     if(Check_Predicate(p_state, is_cond, be_zero, idx_rc))
7     {
8        int16_t  ra = C6XSC5_TO_S16(constant);
9        uint16_t rb = GET_LSB16(p_state->m_reg[idx_rb]);
10       int32_t  rd = ra * rb;
11       SAVE_REG_RESULT(result, idx_rd, rd);
12    }
13    return 0;
14 }
```

Key Point 6.1 Target ISA Definitions in IR

Target ISA behaviors are provided in LLVM-IR and used to compose IR functions for target VLIW binary code. Multiple ISA definitions are given depending on the operand types of a VLIW instruction, for optimization reasons.

6.2.2 Execute Packet Decoding and Basic Block Construction

The target specific VLIW Packet Decoder shown in Figure 6.1 ②, decodes instructions, their operands and creates in-memory objects for later use in translation. Listing 6.2 gives a code snippet for decoding the `MPYSU` instruction of TI C6x ISA. Lines 8, 9 and 10 create objects for operands and Line 11 creates the decoded instruction object. Each decoded instruction object and its operands are able to return their types as a string, as shown in Table 6.1. These string types are used for constructing function calls and naming ISA definitions. The VLIW Packet Decoder also extracts parallelism from the input instruction stream and creates objects representing execute packets. A simple branch analysis is also performed for marking all of the statically known *branch target* instructions. This information is used as an additional criteria for basic block construction.

Target VLIW basic blocks are formed by starting at either a statically known branch target instruction or when a previous VLIW basic block must terminate due to the presence of a branch instruction. When branch instructions do not have delay slots, as is usually the case with RISC machines, basic blocks are terminated immediately. Contrarily in VLIW machines, a branch instruction requires some delay slots, so the execute packets that lie within the the delay slot range of preceding branch instruction are also included in the VLIW basic block, as they *will* be executed even-if the branch is *taken* (Figure 3.15).

6.2.3 Intermediate Code Generation

The intermediate code generation components shown in Figure 6.1 ③ are based on LLVM framework, which provides classes and methods for generating functions, IR basic blocks, control flows and data processing instructions. Target specific features are encapsulated in decoded instruction objects whereas the target independent algorithms are generic and ask the decoded objects for certain services. For example, ISA call generation, where a function call is placed in a given IR basic block by considering the number and operand types of the decoded instruction. This separation of concerns brings target independence to our

Listing 6.2 Instruction Decoding Example

```
1  C6xOperand *src1, *src2, *dest;
2  ...
3  switch(dec_instr->GetOpcode())
4  {
5    case 0x1E :
6    {
7      /* MPYSU Integer Multiply 16lsb x 16lsb; scst5, xulsb16, sint */
8      src1 = new C6xConstant(true,    5, helper->GetSIntConst1());
9      src2 = new C6xRegister(false, 16, helper->GetCrossBankId(), helper->GetSrc2());
10     dest = new C6xRegister(true,  32, helper->GetDestBankId(),helper->GetDest());
11     dec_instr_eu = new C6xMPYSUInstr(dec_instr, dest, src1, src2);
12   }
13   break;
14   ...
15 }
```

Key Point 6.2 Hand-Written VLIW Packet Decoder

A hand-crafted VLIW packet decoder in object-oriented style is used to maximize code reuse. This induces performance penalty during translation but makes the decoding process compatible with LLVM infrastructure.

translation flow.

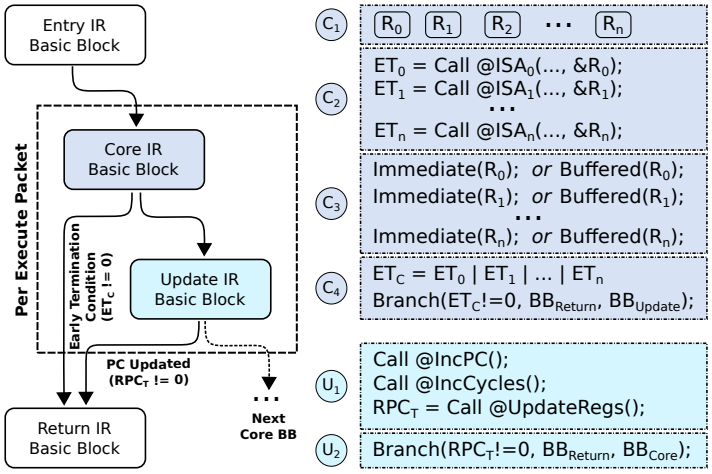


Figure 6.2: Intermediate Code Generation for VLIW Basic Blocks

Figure 6.2 shows the structure of a generated function, containing four types of IR basic blocks *i.e. Entry*, *Return*, *Core* and *Update* types. Each generated function contains one instance of Entry and Return IR basic blocks, whereas the Core and Update IR blocks depend on the number of execute packets in the target VLIW basic block.

Each core basic block contains four groups of instructions, shown as (C_1), (C_2), (C_3) and (C_4). Group (C_1) includes memory allocation instructions for saving *Results* of ISA behavior executions. The number of memory allocations depend on the size of input execute packet and are shown by R_0, R_1, ..., R_n in Figure 6.2. These structures are allocated on the stack of generated function, limiting their visibility and lifetime to the generated IR function. Subsequently during code optimization, we analyze these structures for their scope and

remove unnecessary ones using LLVM optimization facilities.

For group (C2), we generate the actual ISA calls and pass the addresses of corresponding Result structures. Each ISA computes and stores its result(s) in the callers stack memory (Listing 6.1, Line 11). An ISA call also returns a status value shown as ET_0, ET_1, ..., ET_n, which if non-zero indicates an *Early Termination* request from the ISA call, as discussed in Section 3.6.1. The Early Termination request is tested in group (C4) and if true the control returns to *software runtime*

The group (C3) includes either Immediate or Buffered update calls for the ISA execution Results, by considering delay slots associated with the VLIW instructions. This post update scheme brings implicit parallelism to the sequential instructions in IR and avoids the data hazards discussed in Section 3.6.1. Some target instructions can also have side-effects *i.e.* modifications of registers other-than the destination operand(s), as is possible for load and store instructions in TI C6x processors. Group (C3) handles such cases as well, by generating immediate update calls for side-effect results. Instructions in group (C4), evaluate the individual ISA return values to determine if Early Termination Condition is set *i.e.* $ET_C \neq 0$. At runtime, the control flows to the Return basic block for non-zero ET_C values, otherwise it continues to the *Update* IR basic block.

Each *Update* IR basic block is composed of two groups *i.e.* (U1) and (U2). In group (U1), instructions for updating processor state are generated including general purpose registers, program counter and CPU cycles. The register updates deal with the buffered (delayed) updates of registers, including the program counter for branch instructions issued earlier. In cases when the PC is modified during `UpdateRegs`, the new value is saved in RPC_T (Returned Program Counter of Target) for use in group (U2). During simulation if $RPC_T \neq 0$, it indicates that a branch has been taken and control should be returned to the software runtime, which finds the next native function using the RPC_T value and passes control to it. The $RPC_T = 0$ indicates that no branch was taken and the control flows to the next Core IR basic block in the same function, as shown in Figure 6.2. This scheme handles nested branch instructions, as discussed in Section 3.6.1 and Figure 3.15, because the RPC_T value is tested at the end of every *Update* basic block.

Our approach is similar to [NTN06] as we generate a native function for each target basic block but we differ in the respect that we do not use any emulation or ISS based simulation. We keep additional lower level translations for execute packets to achieve faster simulation speed, at the cost of code size. Appendix C provides further details on the code generation algorithms used for VLIW simulation.

Key Point 6.3 Target Independent IR Code Generation

The IR code generation algorithms are target independent as target specific instruction details are encapsulated within the decoded instruction objects.

6.2.4 VLIW Processor State and Circular Buffers

The ISA behavior definitions discussed in Section 6.2.1 includes some additional target specific features. These include the VLIW processor state and a mechanism to support delay slots using circular buffers. The processor state includes the current simulated CPU cycle, register banks and pointers to the circular buffers.

Figure 6.3 shows the structure of circular buffers including head and tail pointers.

MAX_DELAY_SLOTS+1 circular buffers are created during initialization and each buffer contains (MAX_DELAY_SLOTS * MAX_PACKET_SIZE * MAX_INSTR_RESULTS) delay nodes. MAX_DELAY_SLOTS represents the maximum possible delay slots, MAX_PACKET_SIZE is the maximum number of instructions in an execute packet and MAX_INSTR_RESULTS is the maximum number of register results for any instruction.

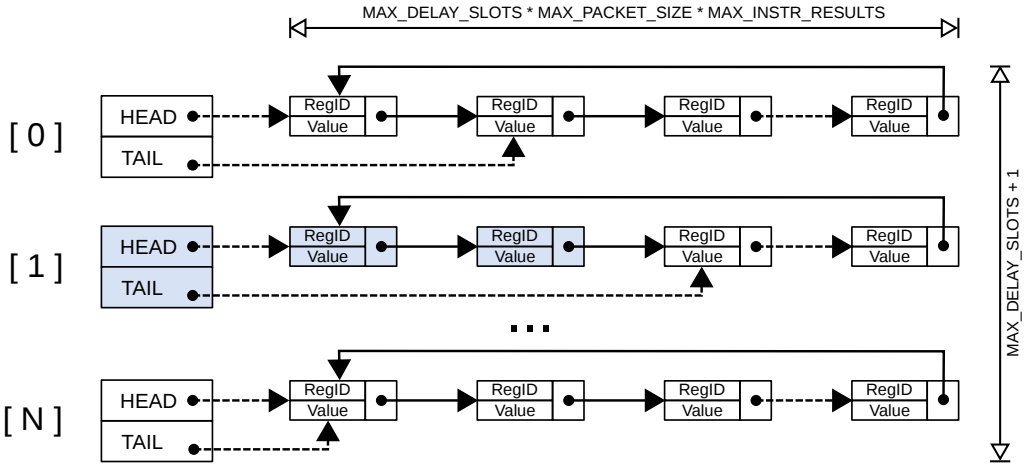


Figure 6.3: Delay Slot Buffers for Register Updates

Each time a VLIW instruction produces a result that requires delay slots, it is saved at the end of a circular buffer, which is selected using the following formula:

$$BufferID = (CurrCPUCycle + DelaySlots + 1) \% (MAX\_DELAY\_SLOTS + 1)$$

On every simulated CPU cycle, the currently active buffer is selected using:

$$BufferID = CurrCPUCycle \% (MAX\_DELAY\_SLOTS + 1)$$

and UpdateRegs function consumes all delayed registers in the current buffer. Figure 6.3 shows the Buffer[1] as active with two delay nodes containing registers to be updated, at the end of current CPU cycle.

6.2.5 Data and Instruction Memory Accesses

The SBT solution is based on the same event-driven simulation framework as described in Chapter 5, where all data and instruction addresses are kept in the original target address space while benefiting from the HAV based address translation. At simulation start, a region of dynamic memory is allocated in host user-space and assigned to KVM for simulating the target address space.

The native simulator generated by the SBT flow accesses data memory using exactly the same addresses as in the original VLIW binary. As the size of generated simulator is larger than the original VLIW binary, it cannot fit within the same memory regions. We use the notion of *Extended Target Address Space*, where we load both the original and translated binaries in the simulated memory, as illustrated in Figure 6.4. For example if a VLIW DSP uses 256MB of memory, we allocate 384MB of memory space in KVM and use the target "physical"

addresses above `0x0FFFFFFF` for the generated simulator.

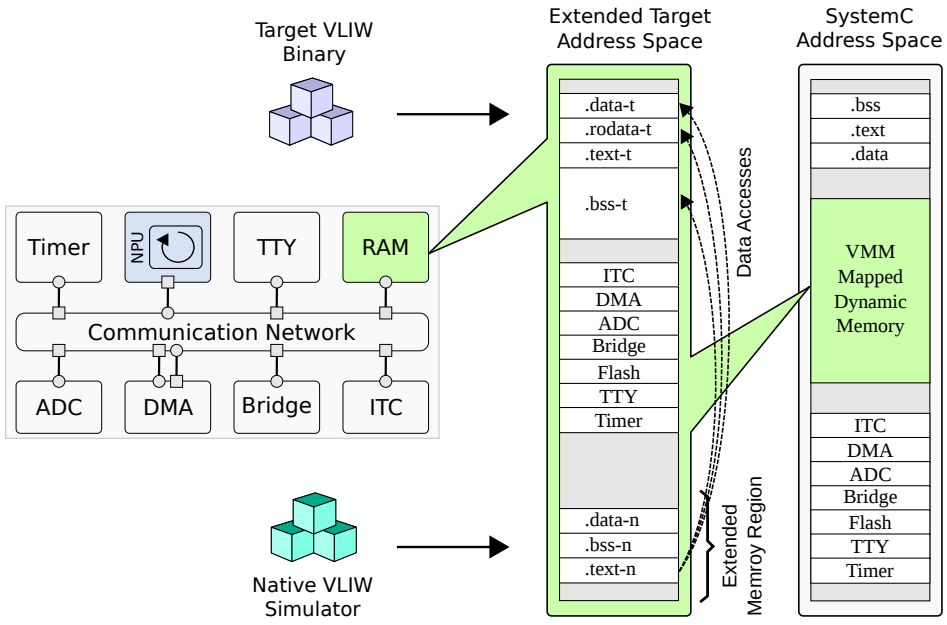


Figure 6.4: Memory Mapping for Statically Generated VLIW equivalent Native Binaries

We load the original VLIW binary and its data sections to their original target memory addresses. This aspect is very similar to UQBT [CVE00], which uses a link-map file generated by the translator to retain the same virtual addresses. The target address preservation for data memory accesses avoids additional computation overhead at runtime [CYH+08].

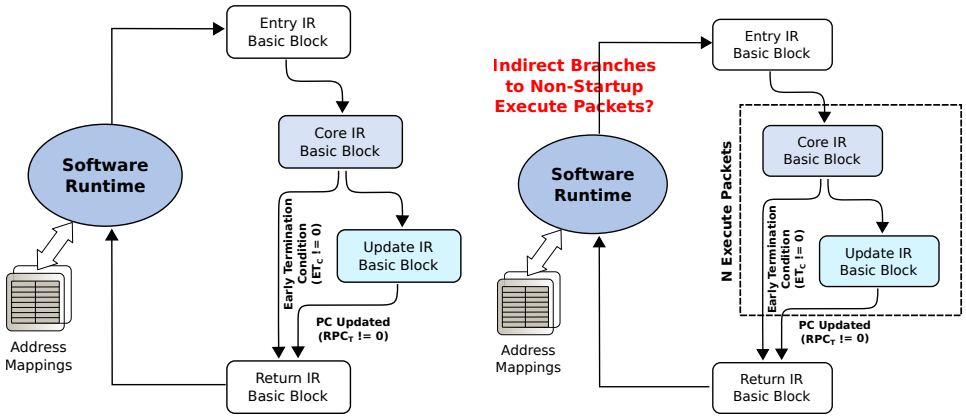
Once a native function completes its execution, it passes control to the software runtime, which looks at the simulated processor state and maps the VLIW program counter to the next native function. A simple option for accomplishing this control transfer is to generate a global map containing target addresses and corresponding native function pointers. This global map can be searched using traditional techniques, such as binary search or a hash function.

Key Point 6.4 Mapping Target Instruction Addresses to Native Functions

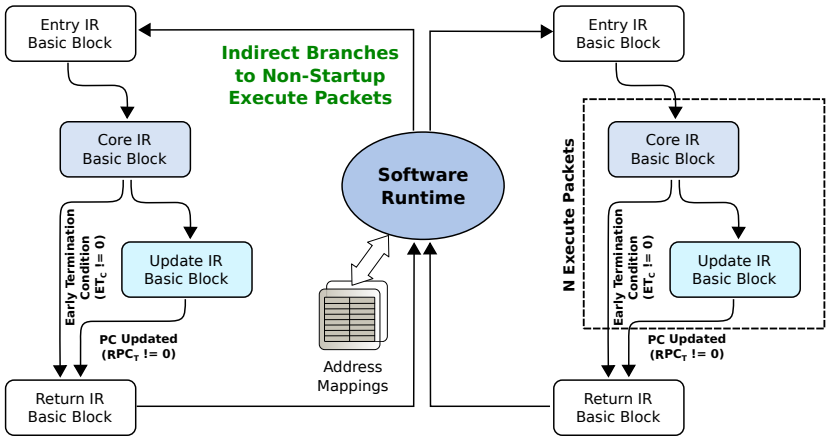
Data memory accesses remain the same as in the original VLIW binary but the generated simulator uses different addresses for instructions. Thus, the SBT for VLIW requires a software runtime and target-to-native address mappings for the generated functions.

6.2.6 Code Generation Modes

Native code can be generated using different modes from the target VLIW binaries. The first option is to generate native code for execute packets only, using the flow Ⓔ in Figure 6.1. This mode suffers from slow simulation speed, as the runtime overhead between native function calls is non-negligible. Using this mode, the generated simulator does not require dynamic translations, as we *know* that all branch instructions including indirect ones will jump to the start of execute packets, as discussed in Section 6.1.1. This code generation mode is shown in Figure 6.5(a).



(a) One IR Basic Block Per Execute Packet (b) One IR Basic Block Per VLIW Basic Block



(c) IR Basic Blocks for VLIW Basic Blocks and Non-Startup Execute Packets

Figure 6.5: Code Generation Modes in Static Translation for VLIW Binaries

To improve the simulation speed, we can generate code for VLIW basic blocks only using the flow Ⓑ in Figure 6.1, but this option is impractical in case of static translation. The presence of indirect branch instructions poses a key problem, as these instructions can jump to execute packets[1] that do not start a VLIW basic block. This implies a jump into middle of the corresponding native function, which is rather impossible. Alternatively we can include dynamic translation or interpretation support in the generated simulator, but this option will make the simulation slower. This code generation mode, although impractical, is shown in Figure 6.5(b).

This brings us to the hybrid translation mode using both Ⓑ and Ⓔ flows of Figure 6.1. This option is practically feasible and interesting from the simulation point of view. This mode is used to generate code for both VLIW basic blocks and execute packets in a mutually exclusive manner *i.e.* in addition to generating code for basic blocks, we also generate code for all non-startup execute packets. The resulting simulator is bigger in size but does not require runtime translations. The simulation speed improves as the simulator remains in basic block mode during most of the runtime and only goes to the execute packet mode when a non-basic block address is encountered, as shown in Figure 6.5(c). Moreover, the

[1] We will refer such execute packets as *non-startup execute packets*.

Generation Mode	Execute Packets (EP)	Basic Blocks (BB)	Hybrid (BB+EP)
Simulation Speed	Slow	Medium	Fast
Simulator Size	Medium	Small	Large
Dynamic Translations	Not Required	Required	Not Required
SystemC Synchronizations	Per EP	Per EP/BB	Per EP/BB
Self Modifying Code	No	Yes	No

Table 6.2: Possible Code Generation Modes for VLIW Software Binaries

hybrid mode is better suited the VLIW context as the minimum translation granularity is an execute packet, compared to an instruction in RISC machines. These translation modes are summarized in Table 6.2.

6.2.7 Optimization and Inlining

LLVM infrastructure provides optimization facilities, known as passes that operate at IR module, function and basic block levels. A module is composed of multiple functions, a function in turn includes one or more basic blocks. Once the intermediate code is available, its quality can be improved using these optimization passes. Some of the key optimization passes include dead code elimination, constant propagation, instruction combining, CFG simplification and instruction inlining.

Listing 6.3 The Dot Product Example in C

```
1  int dot_product(short * m, short * n, int count)
2  {
3      int i;
4      int sum = 0;
5      for (i=0; i < count; i++)
6      {
7          sum = sum + m[i] * n[i];
8      }
9      return(sum);
10 }
```

The proposed translation flow generates some extra code within execute packets, which ensures the functional correctness of simulation. Once the IR is optimized, most of the unnecessary code is removed and we effectively:

- Remove redundant *predicate checks* from un-conditional VLIW ISA definitions.
- Remove dead arguments to ISA calls, such as un-used predicate parameters.
- Merge IR basic blocks that are linked by un-conditional branch instructions.
- Simplify the early termination checks for execute packets containing *fully* executing VLIW instructions *i.e.* instructions without early termination semantics.
- Remove temporary results stored on the generated function's stack and replace them with direct stores to processor state *i.e.* for immediate updates.
- Remove un-necessary ISA calls *e.g.* ISA calls for NOPs.

Listing 6.4 The Dot Product Example in C6x Assembly Language

```
1  /* int dot_product(short * m, short * n, int count) */
2  /* Parameter m in A4, n in B4 and count in A6 */
3  dot_product:
4                      ZERO   A5            /* i = 0 */
5    ||                MV     B4,A8         /* A8 = n */
6                      CMPLT  A5,A6,A1      /* A1 = (i < count) */
7          [!A1]       B      end_for
8    ||    [ A1]       LDH    *+A8[A5],A9   /* A9 = n[i] */
9          [ A1]       LDH    *+A4[A5],A7   /* A7 = m[i] */
10                     ZERO   A3            /* sum = 0   */
11                     NOP    3             /* Branch Delay */
12 for_loop:
13                     ADD    A5,0x01,A5    /* i++ */
14   ||                MPY    A9,A7,A7      /* A7 = n[i] * m[i] */
15                     CMPLT  A5,A6,A1      /* A1 = (i < count) */
16         [ A1]       B      for_loop
17   ||                ADD    A7,A3,A3      /* sum = sum + A7 */
18   ||    [ A1]       LDH    *+A8[A5],A9   /* A9 = n[i] */
19         [ A1]       LDH    *+A4[A5],A7   /* A7 = m[i] */
20                     NOP    4             /* Branch Delay */
21 end_for:
22                     B      B3            /* Return to Caller */
23                     MV     A3,A4         /* A4 = sum */
24                     NOP    4             /* Branch Delay */
```

We illustrate optimization effects using the Dot Product example given in Listing 6.3 and its equivalent C6x assembly code given in Listing 6.4. We use 'C' language style comments to explain instruction in C6x assembler and LLVM-IR code. The execute packet at Lines 13 and 14 includes two instructions *i.e.* an ADD instruction with immediate effect and a MPY instruction with 1 delay slot. The LLVM-IR code corresponding to this execute packet is given in Listing 6.5, which seems overwhelmingly complex. The key reason to this complexity comes from the constraints imposed by VLIW architecture and overheads of translation at execute packet level.

The *getelementptr* (stands for Get Element Pointer) instruction appears very frequently in LLVM-IR and deserves a little explanation. This instruction is used to index into aggregate data types such as arrays and structures. It calculates the appropriate pointer only and *does not* access the corresponding data element *i.e.* it dereferences nothing. During code generation, the getelementptr instructions are translated to constant offsets.

Listing 6.5 includes four IR basic blocks at Lines 2, 5, 8 and 43. Variables for early termination and results are allocated and initialized from Line 9 to Line 15. Lines 17 and 28 invoke the actual ISA behaviors. Immediate and delayed update functions are called at Lines 36 and 38, respectively. The IR basic block at Line 43 updates the simulated VLIW processor state.

To profit from optimizations across multiple ISA calls, we choose to inline function definitions. This improves performance by two-folds, firstly by removing the parameter passing cost of ISA calls and secondly by enabling inter-ISA optimizations. For example, when a register is used as source operand in multiple ISA definitions. Similarly context-sensitive optimizations of ISA definitions can also be enabled *e.g.* an ISA definition can be conditional in one execute packet and un-conditional in another. On the negative side, inlining increases the generated code size, which can become a problem in case of large VLIW binaries.

Listing 6.6 presents an optimized version of the same execute packet. This version

Listing 6.5 LLVM IR Code for Execute Packet at Lines 13 and 14 in Listing 6.4

```
1  define i32 @SimEP_00000020(%struct.C62x_DSPState_t* %p_state) {
2  BB_Entry:
3    br label %BB_00000020_Core
4
5  BB_Return:                    ; preds = %BB_00000020_Update, %BB_00000020_Core
6    ret i32 0
7
8  BB_00000020_Core:             ; preds = %BB_Entry
9    %0 = alloca i32, align 8    /* Allocate Early Termination Variable */
10   store i32 0, i32* %0        /* Initialize it with Zero  */
11   %1 = alloca i32, align 8    /* Allocate Temporary Variable for ISA Return Values */
12   %instr_results = alloca %struct.C62x_Result_t, i32 2, align 8       /* 2 Results */
13   %2 = getelementptr %struct.C62x_Result_t* %instr_results, i32 0
14   %3 = getelementptr %struct.C62x_Result_t* %2, i32 0, i32 0
15   store i32 0, i32* %3        /* Initialize Result 1 with Zero */
16
17   %rvalADD = call i32 @C62xADD_SR32_UC5_SR32(%struct.C62x_DSPState_t* %p_state, i8 0,↩
         i8 0, i16 0, i16 5, i32 1, i16 5, i8 0, %struct.C62x_Result_t* %2)
18
19   store i32 %rvalADD, i32* %1 /* Update Return ISA Value */
20   %4 = load i32* %1           /* Get the Last Return ISA Value */
21   %5 = load i32* %0           /* Get the Early Termination Value */
22   %6 = or i32 %5, %4          /* Calculate Early Termination Flag */
23   store i32 %6, i32* %0       /* Update Early Termination Variable */
24   %7 = getelementptr %struct.C62x_Result_t* %instr_results, i32 1
25   %8 = getelementptr %struct.C62x_Result_t* %7, i32 0, i32 0
26   store i32 0, i32* %8        /* Initialize Result 2 with Zero */
27
28   %rvalMPY = call i32 @C62xMPY_SR16_SR16_SR32(%struct.C62x_DSPState_t* %p_state, i8 ↩
         0, i8 0, i16 0, i16 9, i16 7, i16 7, i8 1, %struct.C62x_Result_t* %7)
29
30   store i32 %rvalMPY, i32* %1 /* Update Return ISA Value */
31   %9 = load i32* %1           /* Get the Last Return ISA Value */
32   %10 = load i32* %0          /* Get the Early Termination Value */
33   %11 = or i32 %10, %9        /* Calculate Early Termination Flag */
34   store i32 %11, i32* %0      /* Update Early Termination Variable */
35   %12 = getelementptr %struct.C62x_Result_t* %instr_results, i32 0
36   %13 = call i32 @Update_Immediate(%struct.C62x_DSPState_t* %p_state, %struct.↩
         C62x_Result_t* %12)
37   %14 = getelementptr %struct.C62x_Result_t* %instr_results, i32 1
38   %15 = call i32 @EnQ_Delay_Result(%struct.C62x_DSPState_t* %p_state, %struct.↩
         C62x_Result_t* %14, i8 1) /* 1 Delay Slot for MPY */
39   %16 = load i32* %0          /* Load Early Termination Variable */
40   %17 = icmp ne i32 %16, 0    /* Check if Early Termination Variable is Set ? */
41   br i1 %17, label %BB_Return, label %BB_00000020_Update
42
43 BB_00000020_Update:           ; preds = %BB_00000020_Core
44   call void @Update_PC(%struct.C62x_DSPState_t* %p_state, i32 8)
45   call void @Inc_DSP_Cycles(%struct.C62x_DSPState_t* %p_state)
46   %Upda = call i32 @Update_Registers(%struct.C62x_DSPState_t* %p_state)
47   br label %BB_Return
48 }
```

includes only one basic block at Line 2, as un-necessary control flows have been removed. Input registers for MPY instruction are loaded and converted to 16-bits from Line 3 to Line 10. Line 12 performs the actual multiplication. LLVM-IR instructions from Line 19 to Line 32 are required for delay slot handling of MPY instruction. In contrast only three instruction are necessary for simulating the ADD instruction *i.e.* Line 15 gets the register from processor state, Line 16 performs the addition and Line 17 immediately updates the result. Instructions between Line 35 and Line 40 update the simulated program counter and CPU cycles. As

Listing 6.6 Optimized LLVM IR Code for Execute Packet at Lines 13 and 14 in Listing 6.4

```
1  define i32 @SimEP_00000020(%struct.C62x_DSPState_t* nocapture %p_state) nounwind {
2  bb3.i8:
3    %p_state.idx2 = getelementptr %struct.C62x_DSPState_t* %p_state, i32 0, i32 1, i32 ↩
         7
4    %p_state.idx2.val = load i32* %p_state.idx2, align 4  /* Get A7 Register */
5    %p_state.idx3 = getelementptr %struct.C62x_DSPState_t* %p_state, i32 0, i32 1, i32 ↩
         9
6    %p_state.idx3.val = load i32* %p_state.idx3, align 4  /* Get A9 Register */
7    %sext.i = shl i32 %p_state.idx3.val, 16               /* Convert A9 to 16 bits */
8    %0 = ashr i32 %sext.i, 16
9    %sext3.i = shl i32 %p_state.idx2.val, 16              /* Convert A7 to 16 bits */
10   %1 = ashr i32 %sext3.i, 16
11
12   %2 = mul nsw i32 %0, %1                               /* MPY     A9,A7,A7 */
13
14   %p_state.idx = getelementptr %struct.C62x_DSPState_t* %p_state, i32 0, i32 1, i32 5
15   %p_state.idx.val = load i32* %p_state.idx, align 4    /* Get A5 Register */
16   %3 = add nsw i32 %p_state.idx.val, 1                  /* ADD     A5,0x01,A5 */
17   store i32 %3, i32* %p_state.idx, align 4              /* Update Immediately */
18
19   %4 = getelementptr inbounds %struct.C62x_DSPState_t* %p_state, i32 0, i32 0
20   %5 = load i64* %4, align 4            /* Load Simulated Cycles */
21   %6 = add i64 %5, 2                    /* MPY Delay Slots+1 to Get Delay Buffer */
22   %7 = urem i64 %6, 6                   /* Calculate Destination Slot */
23   %8 = trunc i64 %7 to i32              /* Convert to 32 bits */
24   %9 = getelementptr inbounds %struct.C62x_DSPState_t* %p_state, i32 0, i32 2, i32 ↩
         %8, i32 1                         /* Get Delay Buffer, 1 Slot Ahead from Now */
25   %10 = load %struct.C62x_Delay_Node_t** %9, align 4   /* Get the Tail Delay Node */
26   %11 = getelementptr inbounds %struct.C62x_Delay_Node_t* %10, i32 0, i32 0
27   store i16 7, i16* %11, align 4                       /* Store Register ID (A7) */
28   %12 = getelementptr inbounds %struct.C62x_Delay_Node_t* %10, i32 0, i32 1
29   store i32 %2, i32* %12, align 4                      /* Store Register Value */
30   %13 = getelementptr inbounds %struct.C62x_Delay_Node_t* %10, i32 0, i32 2
31   %14 = load %struct.C62x_Delay_Node_t** %13, align 4  /* Get Next Delay Node */
32   store %struct.C62x_Delay_Node_t* %14, %struct.C62x_Delay_Node_t** %9, align 4
33
34   %15 = getelementptr inbounds %struct.C62x_DSPState_t* %p_state, i32 0, i32 1, i32 ↩
         47
35   %16 = load i32* %15, align 4          /* Load Simulated PC */
36   %17 = add i32 %16, 8                  /* Increment PC by 8 i.e. for ADD and MPY */
37   store i32 %17, i32* %15, align 4      /* Update PC */
38   %18 = load i64* %4, align 4           /* Load Simulated Cycles */
39   %19 = add i64 %18, 1                  /* Increment by 1 */
40   store i64 %19, i64* %4, align 4       /* Update Cycles  */
41   tail call fastcc void @Update_Registers(%struct.C62x_DSPState_t* %p_state)
42   ret i32 0
43 }
```

none of the instructions in this execute packet have early termination behavior, all such instructions have been removed during the optimization process, which were present in Listing 6.5.

One can clearly see that simulating a VLIW processor on a scalar machine poses many overheads. Especially the delay slot handling and simulated processor state management. Delay slot handling is necessary in order to avoid the data hazards. Processor state management is done once per execute packet, so it is independent of the number of instructions in the execute packet. Thus, optimized VLIW binaries simulate faster using our SBT proposed technique, as compared to un-optimized ones. To further improve the simulation speed, we recommend using the hybrid translation mode, as discussed in

Section 6.2.6. For example, all execute packets from Line 14 to Line 20 in Listing 6.4, lie within a single VLIW basic block as per our VLIW basic blocking criteria, defined in Section 6.2.2. Chapter 7 will present results highlighting the performance benefits of hybrid translations as compared to execute packet only translations.

6.3 Conclusions and Limitations

In this chapter we introduced a static binary translation flow for VLIW processor simulation on HAV based native platforms. The proposed flow is based on a set of algorithms that are generic and retargetable in nature. Thus, the translation flow can be implemented using any compilation infrastructure.

Few limitations of the proposed technique include:

- Redundancy in the generated code in hybrid translation mode, as the proposed solution is completely static. To avoid redundancy, execute packet only translation mode can be used.
- The current solution is semi-automatic as we use hand-written instruction decoder and ISA behavior definitions. Automation of the decoder and ISA definition generation is envisaged.
- Support for self-modifying code is not available in the proposed static translation flow.

And key benefits of the proposed technique are:

- Static translation of complex VLIW binaries to generate optimized native code, requiring no runtime translation or interpretation support, thanks to our HAV based native simulation platform.
- A retargetable approach based on an LLVM-IR, with the ability to generate code for different HAV based platforms.
- A source free approach making it interesting for closed-source VLIW libraries and application simulation.
- Well suited to VLIW architectures, as the minimal granularity of translation is an execute packet for VLIW machines. Nevertheless, this solution can also be applied to RISC architectures.
- Access to final code allows us to generate accurate software annotations during translation for performance estimation and synchronizations with hardware models.
- Data memory accesses are performed using the exact target addresses, making the solution interesting for data cache modeling. Target instruction addresses are available at VLIW basic block and execute packet boundaries, thus, a coarse-grained instruction cache modeling is also feasible.

Next chapter will present experimental results and discussion to validate the contributions of this thesis.

Anyone who has never made a mistake has never tried anything new.

Albert Einstein

7 Experimentation and Results

This chapter presents the necessary experimental results, both in quantitative and qualitative terms, to validate the contributions of this thesis. Initially, a short description of the software and hardware environments, as well as reference benchmarks will be described. Experimental results for mono-processor *System-on-Chip* (**SoC**) will be discussed in detail, to demonstrate the pros and cons of our *Hardware-Assisted Virtualization* (**HAV**) based native simulation solution. Multi-processor and hybrid simulation platform results will be shown, to validate their technical feasibility. We will conclude this chapter by presenting some results for *Static Binary Translation* (**SBT**) of *Very Long Instruction Word* (**VLIW**) software binaries and a discussion on the benefits and limitations of the proposed techniques.

7.1 Software Environment and Benchmarks

Most of the applications used in the experimentation are based on the *APplication Elements for System-on-chips* (**APES**) environment [GP09]. The APES environment provides a set of software components for building embedded software stacks for multiprocessor architectures. These components have well-defined interfaces and provide dedicated OS services. From native simulation perspective, two key elements from the APES environment are considered *i.e.* the DNA Operating System and the HAL layer.

The DNA Operating System has been conceived to target embedded multiprocessor systems. It offers POSIX compatible multi-threading support, dynamic memory management, inter-process communication and synchronization mechanisms. DNA-OS offers all the essential services, with low impact on performance and memory footprint, and provides an easy way to be portable across multiple target platforms, thanks to its HAL API interface. As opposed to [Ger09], we do not make any modifications to the DNA-OS. This becomes possible because the entire software stack is compiled for host machine ISA, but it is executed inside the virtual machine environment provided by KVM *i.e.* in target address space. Thus the software does not have to deal with the dynamically allocated addresses for simulation on native machine and remains un-modified.

The HAL layer serves as a unique interface between software and hardware components, and needs to be re-implemented for porting DNA-OS to a different architecture *i.e.* x86 architecture in our case. The HAL interface of layer is a great incentive from native simulation perspective, as it the only place where hardware specific details are involved and rest of the software stack is independent from the underlying hardware. Moreover the number HAL APIs that need to be implemented for DNA-OS is low (around 30) as compared to complex Operating Systems such as Linux, which would require hundreds of such APIs. Table 7.1 provides the list of HAL API that have been implemented for host machines to enable our solution.

DNA-OS does not support MMU thus all memory references are performed in target "physical" address space and KVM has to provide *Guest Physical Address* (**GPA**) to *Host Physical Address* (**HPA**) translations only, as discussed in Section 5.1.2. Listing 7.1 provides sample implementations for 32-bit memory accesses, which are inline macro definitions and use physical addresses for memory references.

Listing 7.1 Memory Read/Write Accesses using Guest Physical Addresses

```
1  #define CPU_READ_INT32(phy_addr, value)                                    \
2     (value) = *(volatile uint32_t *)(phy_addr)
3
4  #define CPU_WRITE_INT32(phy_addr, value)                                   \
5     *((volatile uint32_t *)(phy_addr)) = (value)
```

Using our solution, the software stack can use statically defined hardware peripherals addresses and these symbols are defined at compile-time, using a linker script. An example linker script is given in Listing 7.2, where DNA-OS specific configurations are defined from Line 7 to Line 20, such as the number of device drivers, required file-systems, stack and heap sizes. A few HAL specific symbols are shown between Line 21 and Line 33. These include application entry point, system stack address, platform debug port, few device register addresses and configuration parameters. The linker script is very similar to the one used for actual target software stack. Following sections describe the set of applications that we will use for our experiments.

7.1.1 MiBench Suite

The advantage of a native simulator is its high simulation speed as compared to cycle/bit accurate simulators. In order to evaluate the performance of our native simulation approach, we use the MiBench [GRE+01] benchmark suite, which includes several classes of embedded applications such as *Automotive/Industrial control*, *Network*, *Security*, *Consumer Electronics*, *Office Automation* and *Telecommunication*. We have selected a set of 12 applications from MiBench with different computation and I/O access properties to characterize the simulation performance of our solution. These applications will be used in mono-processor experiments.

7.1.2 Parallel Motion-JPEG

Motion-JPEG (MJPEG) is a multimedia format where each video frame is encoded independently as a JPEG image. Each video frame can be decoded likewise, without requiring inter-frame decoding information. The Parallel Motion-JPEG application is composed of three types of tasks:

Category	HAL Function	Description
Cache Management	`CPU_CACHE_INVALIDATE()` `CPU_CACHE_SYNC()`	Cache Invalidation and Flushing. (If Modeled)

Context	`CPU_CONTEXT_INIT()`
	`CPU_CONTEXT_LOAD()`
	`CPU_CONTEXT_SAVE()`
		Context management for supporting multi-tasking applications.

Endianness	`PLATFORM_ENDIANNESS`
	`CPU_ENDIANNESS`
	`CPU_DATA_IS_BIG_ENDIAN()`
	`CPU_DATA_IS_LITTLE_ENDIAN()`
	`CPU_PLATFORM_TO_CPU()`
	`CPU_CPU_TO_PLATFORM()`
		Platform and CPU Endianness and Conversion APIs.

Memory and I/O Support	`CPU_[READ|WRITE]()`
	`CPU_IO_[READ|WRITE]()`
	`CPU_UNCACHED_[READ|WRITE]()`
	`CPU_VECTOR_[READ|WRITE]()`
	`CPU_VECTOR_TRANSFER()`
		Memory and I/O Device accesses; Each API provides multiple modes of operation (8, 16, 32, 64, SFLOAT, DFLOAT)

Multiprocessor Support	`CPU_MP_[ID|COUNT]()`
	`CPU_MP_[WAIT|PROCEED]()`
	`CPU_MP_SEND_IPI()`
		Multiprocessor support including identification and interprocessor synchronizations.

Power	`CPU_POWER_WAKE_ON_INTERRUPT()`	Power Management (Idle Thread)
Synchronization	`CPU_TEST_AND_SET()` `CPU_COMPARE_AND_SWAP()`	Synchronization primitives for providing atomic accesses.

Traps	`CPU_TRAP_COUNT()`
	`CPU_TRAP_ATTACH_[ESR|ISR]()`
	`CPU_TRAP_MASK_AND_BACKUP()`
	`CPU_TRAP_RESTORE()`
	`CPU_TRAP_[ENABLE|DISABLE]()`
		Interrupts and Exception handling (Attaching/Enabling/Disabling/Masking)

Timing	`CPU_TIMER_[GET|SET]()` `CPU_TIMER_CANCEL()`	Time Management.

Table 7.1: HAL API Functions for DNA-OS

- DISPATCHER: Parses the encoded input video stream from a file or a device model and sends it to the decoder tasks using software channels.

- DECODER: Gets input video frames from the dispatcher and decodes them by applying the *Inverse Discrete Cosine Transform* (**IDCT**) to each of the macroblocks. The number of decoder tasks is configurable at application level and it is used to increase the software parallelism.

- SERIALIZER: Takes the decoded video frames from one or more decoders and sends them to the Framebuffer device model for visualization.

Figure 7.1 shows the functional decomposition of Parallel-MJPEG application. All of the application tasks are implemented using software threads and execute on top of DNA-OS.

Listing 7.2 An Example Linker Script with DNA-OS and Hardware Device Configurations

```
1  SECTIONS
2  {
3      .init 0x0100000: { stext = .; *(.reset) }                          > mem :text
4      .text ALIGN(0x8): { *(.text) *(.text.*) }                          > mem :text
5      .data ALIGN(0x8) : { *(.data*) *(.glue_7*) *(.eh_frame*) }         > mem :data
6
7      .os_config ALIGN(0x8): {
8          OS_N_DRIVERS = .; LONG(0x5)
9          OS_DRIVERS_LIST = .; LONG(soclib_platform_module) LONG(rdv_module)
10                             LONG(fdaccess_module) LONG(soclib_fb_module)
11                             LONG(tg_module)
12         OS_N_FILESYSTEMS = .; LONG(0x2)
13         OS_FILESYSTEMS_LIST = .; LONG(devfs_module) LONG(rootfs_module)
14
15         OS_THREAD_STACK_SIZE = .; LONG(0x8000)
16         OS_KERNEL_HEAP_ADDRESS = .; LONG(ADDR(.kheap))
17         OS_KERNEL_HEAP_SIZE = .; LONG(0x1000000)
18         OS_USER_HEAP_ADDRESS = .; LONG(ADDR(.uheap))
19         ...
20     }                                                                  > mem :data
21     .hal ALIGN(0x8): {
22         APP_ENTRY_POINT = .; LONG(_main)
23         CPU_SYS_STACK_ADDR  = .; LONG(ADDR(.sysstack))
24         CPU_BSS_START = .; LONG(ADDR(.bss))
25         CPU_BSS_END = .; LONG(__hal_bss_end)
26
27         PLATFORM_DEBUG_CHARPORT = .; LONG(0xC0000000)
28         SOCLIB_FB_NDEV = .; LONG(0x1)
29         SOCLIB_FB_DEVICES = .; LONG(256) LONG(144) LONG(0XC4001000)
30         SOCLIB_FDACCESS_NDEV = .; LONG(0x1)
31         SOCLIB_FDACCESS_DEVICES = .; LONG(2) LONG(5) LONG(0xC3000000)
32         ...
33     }                                                                  > mem :data
34
35     .rodata ALIGN(16): { *(.rodata) }                                  > mem :data
36     .sysstack ALIGN(0x8) + 0x10000 : { }                               > mem :stack
37     .bss ALIGN(16): { *(.bss) *(.rel*) *(COMMON) __hal_bss_end = .;}   > mem :bss
38     .kheap ALIGN(0x8) : {}                                             > mem :heap
39     .uheap ALIGN(0x8) + 0x1000000: {_end = .;}                         > mem :heap
40     edata = .;
41 }
```

7.1.3 Audio Filter

The Audio Filter[1] application is frequently used in the communication domain. This application will be used in hybrid simulation context to demonstrate the use of shared memories between different types of simulation technologies. In hybrid simulation platform, two software stacks will be used *i.e.* Natively compiled Audio Filter application on DNA-OS and Cross-compiled DSP driver on Linux. The DSP driver sends an audio frame to the Audio Filter application using shared memory. The Audio Filter example performs Fourier analysis and synthesis to perform the filtering on shared memory data. Both software stacks use a global shared memory lock for synchronization purposes.

[1] This application was provided by Thales Communications.

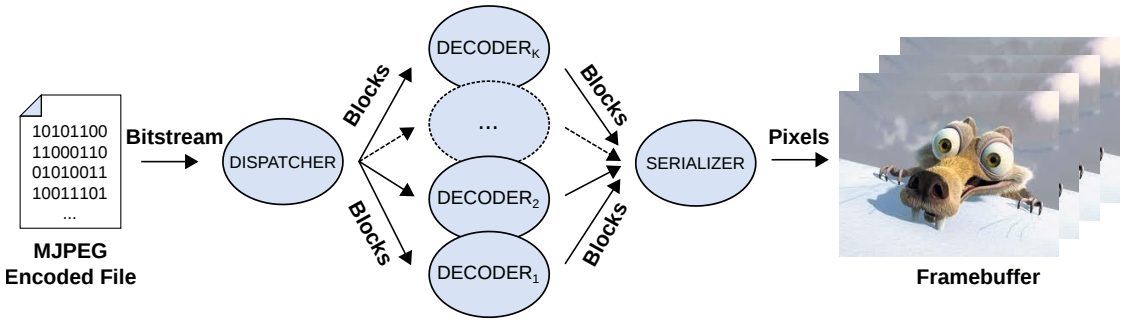


Figure 7.1: Functional Decomposition of Parallel-MJPEG Application

7.1.4 DSP Kernels

A set of DSP kernels has been selected from the PolyBench [Pol] suite for testing the VLIW simulation using the SBT technique proposed in Chapter 6. These include benchmarks for *Linear Algebra*, *Data Mining*, *2-D Image Processing* and *2-D Stencil* computations. Additionally compute and control dominant kernel are used to show the effect on simulation performance of reference simulators and the proposed SBT generated simulators. Table 7.2 gives the list of selected DSP kernels along with a short description.

DSP Kernel	Short Description
ADI	Alternating Direction Implicit solver.
Correlation	Correlation computation.
Covariance	Covariance computation.
Doitgen	Multi-resolution analysis kernel (MADNESS).
FDTD-2D	2-D Finite Different Time Domain Kernel.
IDCT	Inverse Discrete Cosine Transform.
Jacobi-2D	2-D Jacobi stencil computation.
LU	LU decomposition.
MVT	Matrix Vector product and Transpose.
Reg-Detect	2-D Image processing.

Table 7.2: Selected DSP Kernels for VLIW Simulation

7.2 Hardware Environment and Reference Platforms

The hardware models used in experimentation are based on the components taken from the RABBITS framework [RAB]. These components have been principally developed for use in QEMU based simulation platforms. Our KVM based solution can also use these components, as the software execution takes place in target address space, thus exactly matching the binary translation based simulation address spaces.

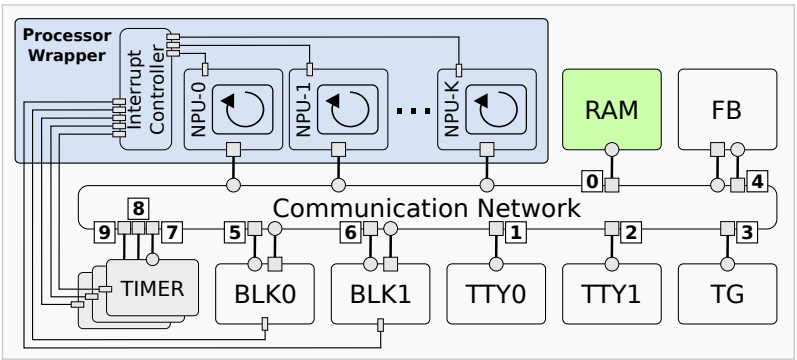
The hardware components define two types of device models *i.e.* master and slave devices. Each master device maintains its address decoding table and can initiate transactions on the

communication component targeting a specific slave device. Slave devices react to requests from master devices and respond accordingly, either by reading or writing to device registers, and sending the response packets to the transaction initiating master device. The slave devices are not allowed to initiate transactions on the communication network. A device model can be defined as a master and slave device, and in such case it exports interfaces for both types of device models.

7.2.1 Native MPSoC Simulation Platform

The native MPSoC platform will serve as our principle experimental testbed and most of the experiments will use it for different types of software applications. Firstly this platform will be used to demonstrate the software execution in target address space. The use of arbitrary addresses for device models will highlight that overlapping addresses can be used in our HAV based technique. In the case of mono-processor experiments, this platform will be used to demonstrate the differences in simulation performance of compute and I/O intensive applications. For multi-processor experiments, we will use the Parallel-MJPEG application and highlight the effect of number of processing elements on simulation performance. Lastly, this platform will be used for simulating VLIW DSP kernels, listed in Section 7.1.4. The SBT technique proposed in Chapter 6, will be used for generating native code from cross-compiled software kernels.

The platform shown in Figure 7.2 includes two Block Devices (BLK0 and BLK1)(Master and Slave), two terminals (TTY)(Slaves), a Traffic Generator (TG) (Slave) and a Framebuffer (FB) (Master and Slave) device models. Block devices are required for simulating the MiBench applications, as these applications require file I/O support. The traffic generator provides input data for the Parallel-MJPEG application and the Framebuffer model is used for visualizing the decoded images. When simulating VLIW software binaries, multiple TTY devices provide support for distinguishing between debug and standard output results. This enables us to compare our simulation results with reference cycle-accurate platforms. The device address mapping shown on the right side of the Figure 7.2 is shared by all master devices in the platform *i.e.* All CPUs, Block Devices and Framebuffer models.



Device	Addr. Range	Port
RAM	0x00000000 0x0FFFFFFF	0
FB	0xC4000000 0xC40FFFFF	4
TTY0	0xC0000000 0xC00001FF	1
TTY1	0xC0000200 0xC00003FF	2
TG	0xC3000000 0xC3000FFF	3
BLK0	0xC6000000 0xC60FFFFF	5
BLK1	0xC6500000 0xC65FFFFF	6
Timer0	0xC1000000 0xC100000F	7
Timer1	0xC1000010 0xC100001F	8
Timer2	0xC1000020 0xC100002F	9

Figure 7.2: NaSiK MPSoC Simulation Platform for Native and VLIW Simulation

7.2.2 Hybrid Simulation Platform

Hybrid simulation platform will be used to demonstrate the feasibility of mixed simulation, where different types of processing elements are used on top of the same hardware platform. We will use two different technologies *i.e.* KVM for native and QEMU based ISSes for cross-compiled software execution. The Audio Filter application mentioned in Section 7.1.3 will be executed on top of DNA-OS, and it will receive data for processing from a DSP driver application running on Linux. Principle objective of this experiment is to show the feasibility of shared memory accesses and mixing of different simulation technologies together in a single platform. Figure 7.3 shows the overall design of the platform used for hybrid simulation experiment with multiple memory models, timer devices and terminals.

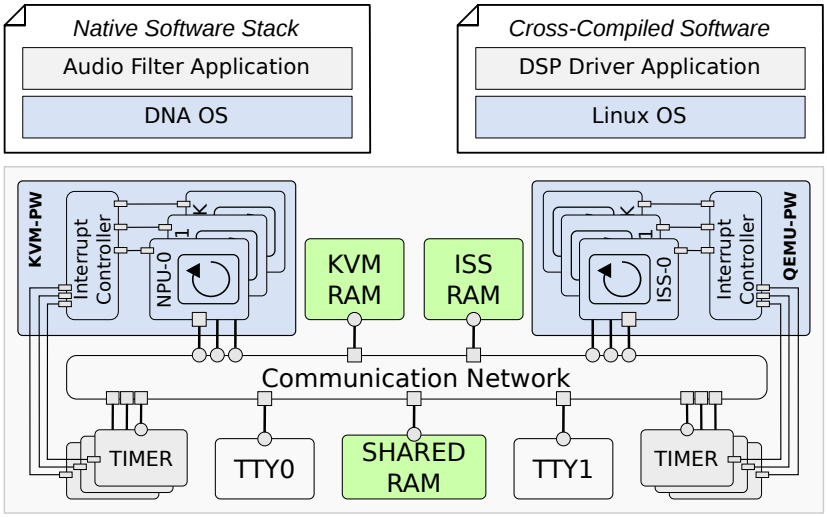


Figure 7.3: Hybrid Simulation Platform using KVM and QEMU Based Processor Models

7.2.3 Reference Platforms and Simulators

The simulation platform proposed in [GHP09] will serve as a principle reference for comparison in native simulation domain. This platform will be referred to as *Native* in all of the experimental results and discussion. The simulation platform proposed in [GFP09], which uses QEMU based ISSes, will be referred to as *Rabbits* in the rest of this chapter. Results from our HAV based native simulation solution will be referred to as *NaSiK*.

The SBT based solution for VLIW machines has been implemented for *Texas Instruments* (**TI**) C6000 Series DSPs. We will use the Full Cycle Accurate (FCA) and Device Functional Cycle Accurate (DFCA) simulators from TI for comparison with simulation of SBT generated native code. These simulators will be referred to as *TI-C6x-FCA* and *TI-C6x-DFCA*, respectively. The TI-C6x-FCA models all of system components with cycle accuracy and TI-C6x-DFCA models the processor with cycle accuracy and rest of system components at functional level.

7.3 Mono-processor Experiments

This section presents experiments based on the platform introduced in Figure 7.2, with a single native processor model. All of these experiments use the MiBench applications

introduced in Section 7.1.1. Key objectives of these experiments are to show that software execution takes place in target address space and the KVM solution is very similar in speed to the traditional native techniques and faster than the DBT techniques.

7.3.1 Software Execution in Target Address Space

The first experiment shows that the software execution takes place in target address space. A single processing element, a RAM model and two I/O devices are shown in Figure 7.4, in order to demonstrate the address usage. The memory model does not show any transactions from the software execution, as all memory accesses are transparent and take place within the guest mode. The RAM model will only show memory accesses when performed by a hardware device model such as a DMA device, or when software accesses to the memory model are mapped using MMIO mechanism. MMIO based memory accesses will be discussed in Section 7.4.2.

I/O device accesses are shown for the terminal and the Framebuffer models in Figure 7.4. When the CPU writes to `0xC0000000`, which is the base address of terminal device, it is shown as `0x00000000` Ⓐ on the terminal's trace log because the offset is zero. Similarly, when the CPU writes to the Framebuffer, it uses the address `0xC4001000`, which appears as `0x00001000` Ⓑ for the same reason. These memory references use addresses that are usually unavailable to the user-mode software. In our solution, we can use such addresses because these are mapped by the MMIO mechanism provided by KVM. Thus, memory address overlapping issue between the host machines and the simulated software can be effectively resolved.

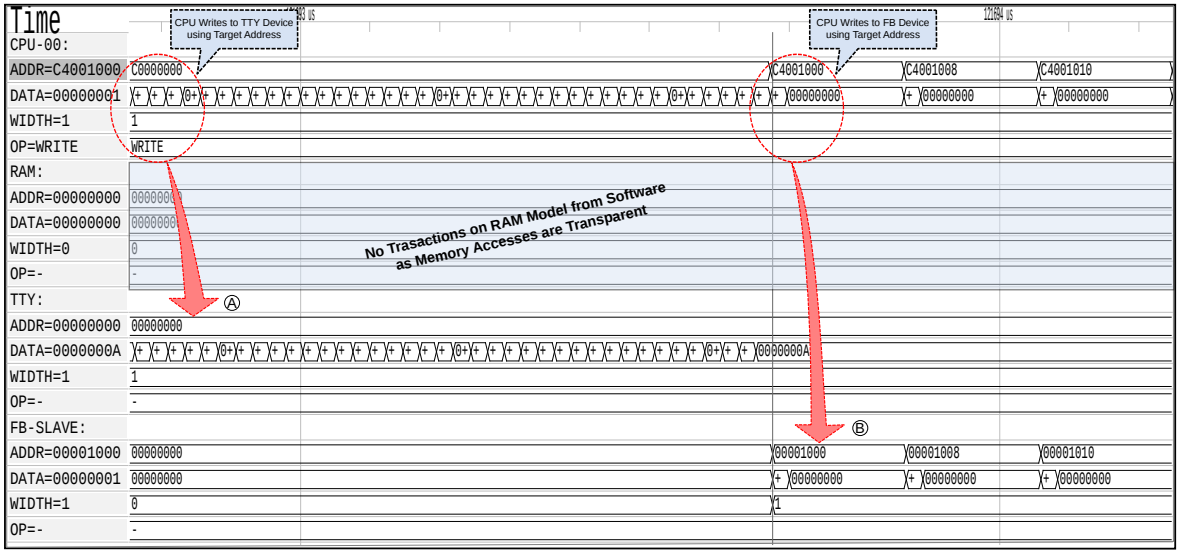


Figure 7.4: Target Memory and I/O Address Space Accesses

7.3.2 Compute *vs.* I/O Intensive Applications

In most of the selected MiBench applications, the I/O time is much larger than the computation time, which makes it difficult to compare the simulation speed of computations with reference platforms. For this reason a semi-hosting based profiling mechanism has been

used to separate computations from I/O accesses. The test applications are executed between 5 to 250 times, in order to arrive at a reasonable computation time. This count is mentioned as XN after each application's name where N denotes the number times an application is executed to obtain the given result.

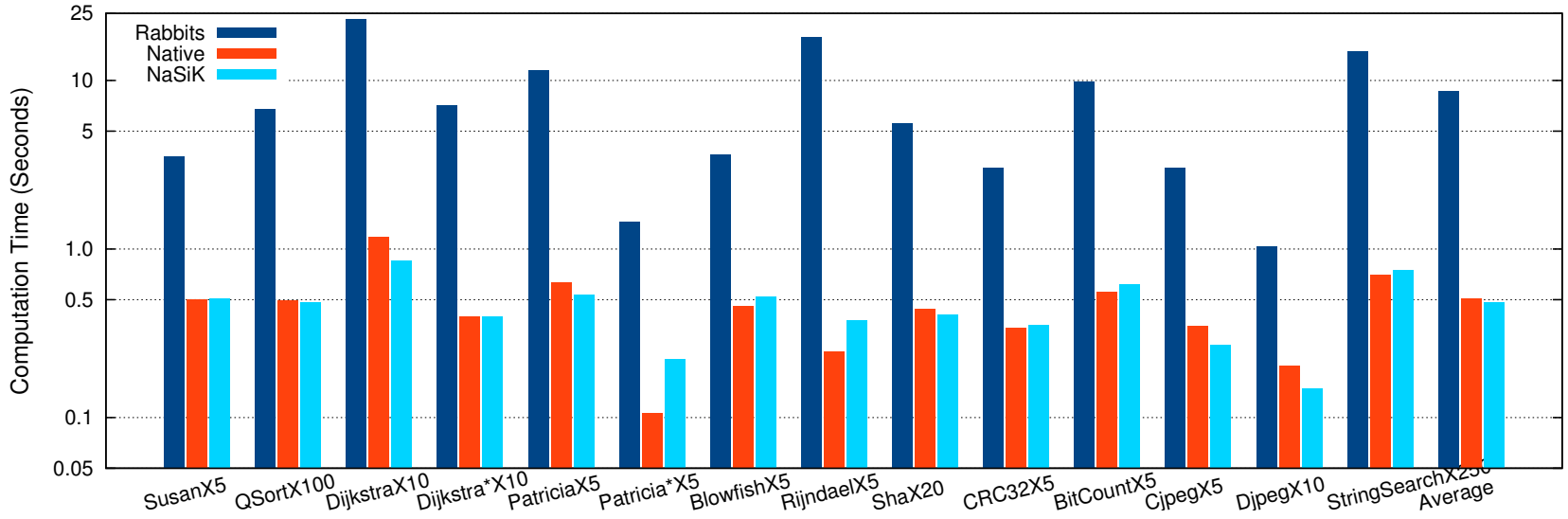
The Figure 7.5(a), shows the simulation performance w.r.t computations. At a glance, one can see that the simulation speed of NaSiK simulation is much faster than Rabbits in all cases, while we achieve almost the same performance level as compared to the previously proposed Native simulation. However, there are a few cases, *e.g.* Dijkstra, Patricia, Cjpeg and Djpeg, where NaSiK simulation performs better than the Native simulation. Further investigations reveal that this speedup is due to the dynamic memory allocations/de-allocations, which require HAL layer support in the Native strategy and this is not the case for NaSiK based native technique. Results for statically allocated MiBench applications have also been shown in Figure 7.5(a), with an asterisk *(*)* symbol to indicate this fact.

Although QEMU is recognized as an efficient dynamic binary translator, it has an obvious performance drawback when compared to native simulation strategies and it is due to the runtime instruction translation and basic block chaining overhead. Table 7.3 summarizes the simulation speedup achieved by our NaSiK simulator as compared to Rabbits and Native strategies. We highlight the best and worst cases for MiBench applications as plotted in Figure 7.5(a), along with the overall averages for all applications except for the static memory versions of Dijkstra and Patricia applications. The noteworthy numbers are 17.91X speedup against the Rabbits simulator and 1.05X speedup against the Native technique.

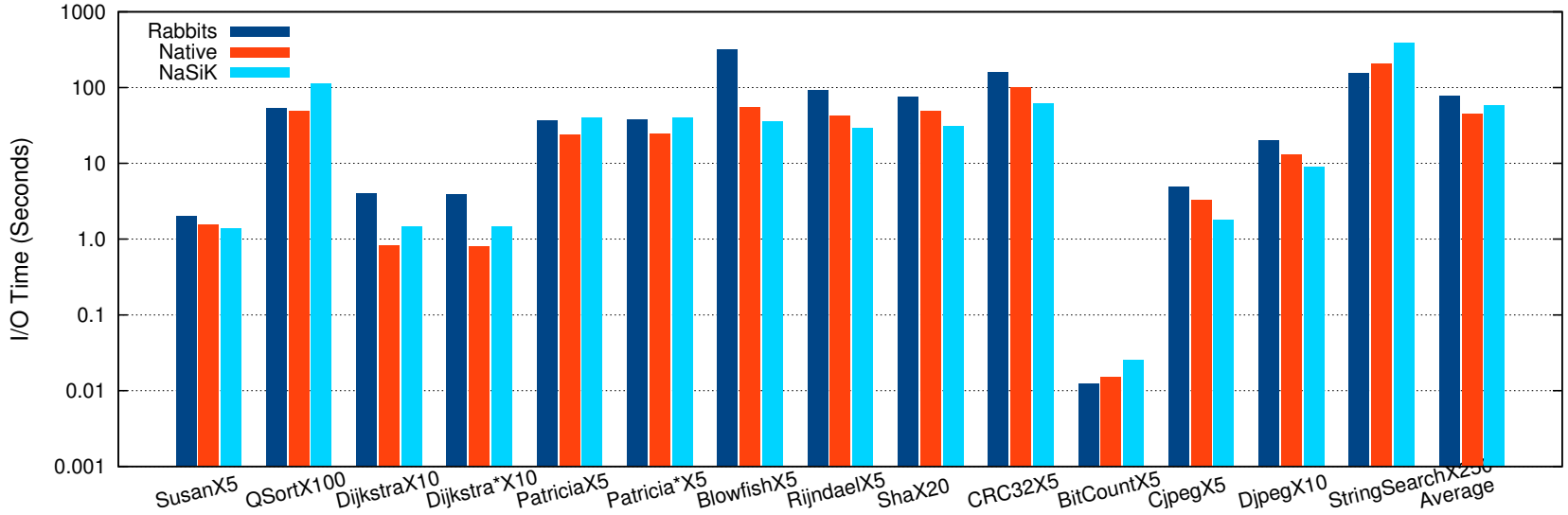
In order to demonstrate that the computation performance of NaSiK and Native platforms is the same, we present the test results of Pi example in Figure 7.6. The accuracy level of Pi calculation is varied, between 1K to 100K decimal digits and computation time is compared with the reference platforms. Results of Native and NaSiK solution are very similar, whereas the Rabbits platform is slower by a constant factor of 14.67X at all accuracy levels. These results agree with the ones shown in the Figure 7.5(a) and Table 7.3.

Simulation Platform	Best-Case	Worst-Case	Total Time
	Rijndael	Djpeg	All Applications
Rabbits	18.081s	1.033s	104.099s
NaSiK	0.376s	0.148s	5.814s
Speedup/Slowdown	48.10X	6.96X	17.91X
	Dijkstra	Rijndael	All Applications
Native	1.185s	0.246s	6.104s
NaSiK	0.846s	0.376s	5.814s
Speedup/Slowdown	1.40X	0.66X	1.05X

Table 7.3: Computation Speed-up in KVM Simulation



(a) Computation Speed Comparison



(b) I/O Speed Comparison

Figure 7.5: Computation and I/O Speed Comparison between Rabbits, Native and NaSiK Platforms

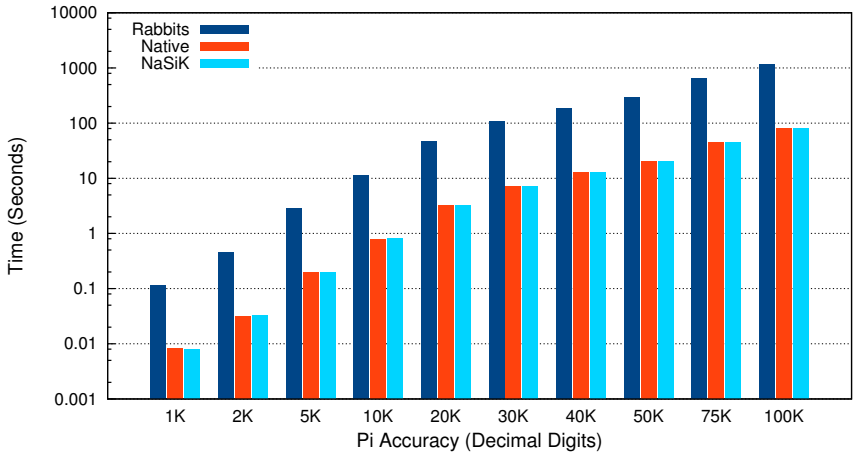


Figure 7.6: Computation Performance of PI Application for Different Accuracy Levels

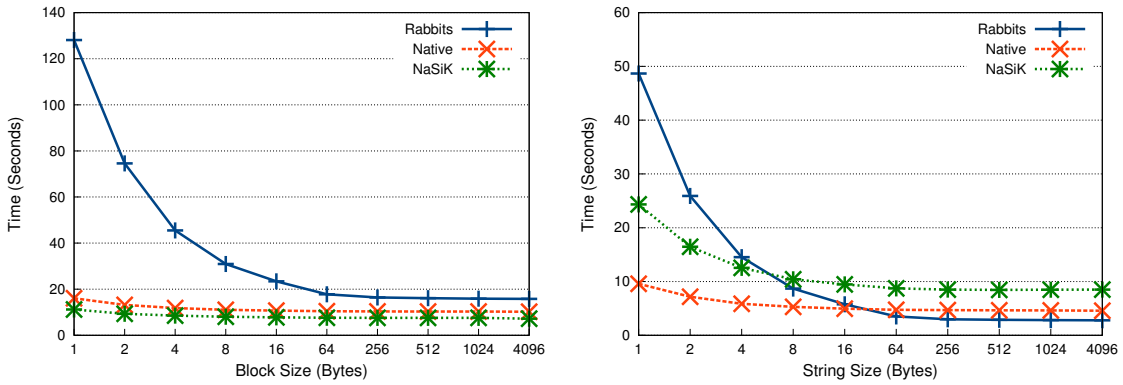
The I/O communications are the real bottleneck in Rabbits, as well as Native simulators. This is particularly true for the MiBench applications, and the situation is further deteriorated in NaSiK, due to the guest-to-host mode switching overhead. Each time the CPU encounters an I/O request, it has to save its current guest mode state and switch to the user mode for fulfilling this request. Rabbits has a similar behavior as well, as it has to stop the execution of translated binaries for the SystemC I/O operations. Moreover the Native simulator has its own bottlenecks, like one of the key issue is its dependency on HAL layer for performing I/O operations that lead to thread switching when accessing a SystemC device. Figure 7.5(b) shows the simulation performance *w.r.t.* I/O accesses for the same set of MiBench applications.

Simulation Platform	Best-Case	Worst-Case	Total Time
	Blowfish	StringSearch	All Applications
Rabbits	314.190s	155.247s	916.112s
NaSiK	35.768s	385.659s	708.708s
Speedup/Slowdown	8.78X	0.40X	1.29X
	Cjpeg	QSort	All Applications
Native	3.271s	48.946s	545.136s
NaSiK	1.801s	113.054s	708.708s
Speedup/Slowdown	1.82X	0.43X	0.77X

Table 7.4: I/O Speedup/Slowdown in KVM Simulation

In order to support and explain the I/O performance results of Figure 7.5(b), we will initially focus on the results of two specialized I/O applications. In the first test, the block size is varied at the application level for copying data from source to destination file (file size = `4 MB`) and compare the performance results of given platforms. Figure 7.7(a) shows that the performance improves, as we increase the block size for all platforms. This improvement is rapid in Rabbits, but it never surpasses the performance of Native and NaSiK platforms.

The improvement remains in the range of 11.45X (Block Size = 1) to 2.10X (Block Size = 1024). This lower performance is linked to the ratio of computation overhead introduced by QEMU and the actual amount of I/O (Block Size) performed. As the block size increases, this ratio decreases and I/O performance improves. Similarly, the I/O performance of Native is in the range of 1.44X (Block Size = 1) to 1.37X (Block Size = 1024). On average, NaSiK shows an improvement of 4.66X against Rabbits and 1.39X against Native solutions for the block I/O test.



(a) Block I/O Performance

(b) Console I/O Performance

Figure 7.7: Block and Console I/O Performance for Rabbits, Native and NaSiK Simulation Platforms

The second test focuses on printing I/O operations where a string output function (fputs in C) is used at application level to print 1MB of characters to the TTY device. Similar to the first test, the string length is varied and its effect determines the platform performance. For example, for string length of 1 byte, the string output function is called 1048576 times, and for string length of 16 bytes the same function is invoked 65536 times. The output is flushed after every function call, so that each platform quits its current operating mode (Guest in NaSiK), and the output is immediately displayed on the terminal screen. Figure 7.7(b) shows that Rabbits surpasses the performance of both NaSiK and Native solutions at string lengths of 8 and 64 bytes respectively. The NaSiK solution ranges between 2.00X faster (String Length = 1) to 0.33X slower (String Length = 4096), when compared to Rabbits. With reference to Native platform, the NaSiK solution always remains slower in the range of 0.39X (String Length = 1) to 0.55X (String Length = 1024). On the average a speedup of 1.02X against Rabbits and a slowdown of 0.48X against Native solution is observed, for the Console I/O test.

MiBench applications use either one or both types of I/O operations, and their combination determines the overall I/O performance of a given application. For example, if an application performs fine grain I/O operations *i.e.* Reads/Writes in characters instead of large chunks of data, the cost of mode switching (especially in NaSiK) can become a dominant factor in the overall I/O cost. This is the case for QSort, Dijkstra, Patricia, BitCount and StringSearch applications.

On the other hand, if an application performs I/O operations using block devices, the operating system ensures that we read a large block of data and quit the Guest Mode only when a new block of data needs to be read or written. The operating system's internal buffers are used to meet the fine grain I/O needs of the application. This type of behavior is seen in

Susan, Blowfish, Rijndael, Sha, CRC32, Cjpeg and Djpeg applications. Table 7.4 summarizes the I/O performance of all applications.

From the results and discussion above, we conclude that I/O communication cost is a performance issue for the KVM based NaSiK platform, but it would be so for any solution that relies on an event driven simulator (even at Transaction Level) to simulate the hardware components.

7.3.3 Software Annotations and Simulation Accuracy

This section discusses the simulation performance of NaSiK simulation platform when software annotations are enabled. The annotation strategy used in these experiments was discussed in Section 5.3. Figure 7.8 shows the results for three different experiments, for the same set of MiBench applications on the NaSiK simulation platform.

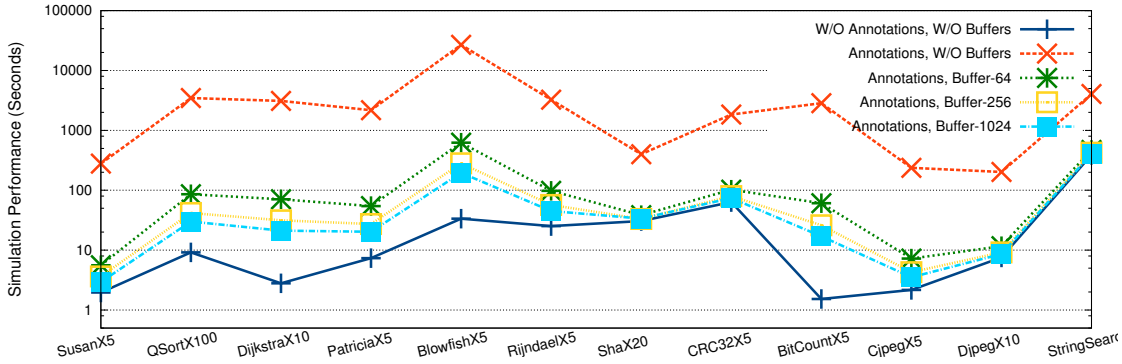


Figure 7.8: NaSiK Simulation Platform with Annotations and Annotation Buffers

In the first experiment, annotations are disabled and these results serve as a reference for comparison with rest of the experiments where annotations are enabled and the annotation buffer size is varied to observe its effect on performance. The annotation buffers are maintained on the software side and guest-to-host mode switch takes place only when a buffer overflows. At this instant the user mode is entered and the SystemC `wait` function is called for all annotations present in the buffer. In this way, the mode switching cost as well as the SystemC synchronization overhead is reduced; however, at the cost of simulation accuracy.

In the second experiment, annotations are enabled and the annotation buffer size is set to zero, forcing NaSiK to quit Guest mode on each annotation call. Results show that the simulation becomes very slow in this case, with worst case slowdown reaching a factor of almost `2000X` and even in the best case, NaSiK simulation is slower by a factor of `10X`. These results highlight the importance of annotation buffering in order to achieve reasonable simulation performance.

In the third experiment, annotation buffers are enabled and the annotation buffer size is tweaked to see its effect. This experiment is repeated for `K` = `64`, `256` and `1024`, where `K` is the annotation buffer size and results of this experiment show improved simulation performance. The slowdown factor is reduced to `2.83X`, `1.77X` and `1.49X`, respectively. Table 7.5 summarizes these results.

Buffer Size	Best-Case	Worst-Case	Average Slowdown
	StringSearch	BitCount	All Applications
0	10.38X	1862.38X	84.56X
64	1.20X	39.69X	2.83X
256	1.08X	17.00X	1.77X
1024	1.04X	11.12X	1.49X

Table 7.5: KVM Simulation Performance With Annotations and Annotation Buffers

Error Type	Best-Case	Worst-Case	Average Error	Absolute Error
	Rijndael	StringSearch	All Applications	
Error Instructions	-0.02%	-16.60%	-3.95%	5.80%
Error CPU-Cycles	+0.55%	-14.06%	-2.77%	5.36%

Table 7.6: KVM Simulation Accuracy Best and Worst Cases

7.3.3.1 Simulation Accuracy

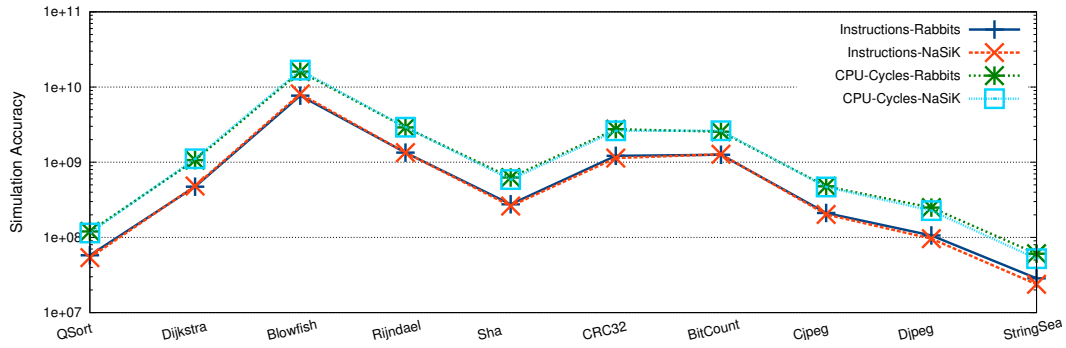
We discuss the accuracy of the used annotation scheme in terms of instruction and processor cycle counts *w.r.t.* the Rabbits platform. The key idea of presenting these results is to demonstrate the fact that NaSiK solution is compatible with all types of software annotations. A subset of previously mentioned MiBench applications has been selected for accuracy comparisons. We were forced to remove Susan and Patricia applications from these results due to limitations in the compilation infrastructure and KVM, this will be discussed later.

To measure accuracy, each application is executed once and the cost of main function is reported, in terms of the number of equivalent *target* instructions executed and the number of *target* CPU cycles consumed by these instructions. Results of these measurements are shown in Figure 7.9(a) where, for certain applications, a slight offset is visible in the plotted results. To further highlight these differences, the percentage error in instructions and cycles counts is given in Figure 7.9(b). Although the accuracy of these annotations is less than ideal, nevertheless these results are usable in early design space exploration stages.

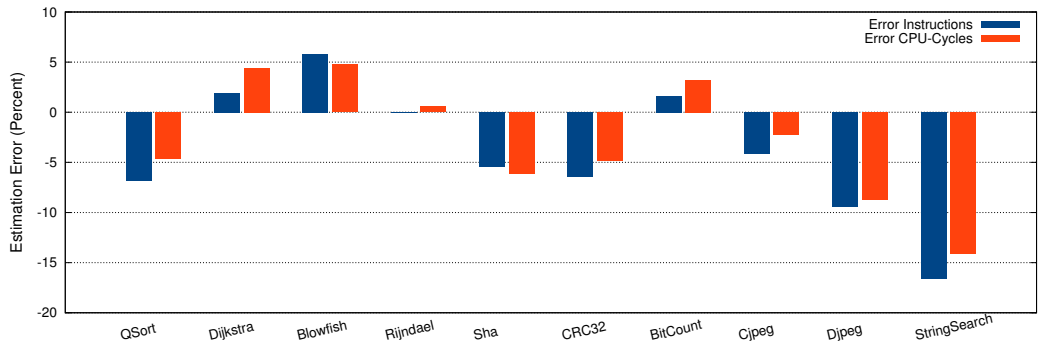
Table 7.6 gives the best and worst cases in performance evaluation. Key figures are given in the last column, which shows that on average fewer number of instructions and CPU cycles are reported. This is due to the absence of annotations in some of the software sources that could not compiled using the LLVM toolchain. The absolute error for the selected set of MiBench applications is about 6%.

7.3.3.2 Sources of Annotation Inaccuracies

As evident from the results, the annotation technique can be quite inaccurate in some cases. There are a few fundamental limitations of the annotation approach used. The first type of errors emerge due the absence of annotations when the given source code is comprised



(a) Instructions and Processor Cycles Accuracy



(b) Percentage Error in Instruction and CPU-Cycle Estimation

Figure 7.9: Instruction and Cycle Accuracy Comparison between Rabbits and NaSiK

of assembly files, commonly found in NewLib and low-level software libraries. As the annotation technique is based on the LLVM IR, we cannot compile and annotate assembly files and there are around `20` such files present in the NewLib sources.

The second type of errors come from limitations of the compilation tools used to perform the annotation. As we make a special use of the LLVM infrastructure for the compilation of software where we initially compile the sources for the given target and annotate the IR, and then recompile this modified representation for the host machine. This dual compilation flow fails in some cases, for example when the source code contains `long double` data types and we are interested in annotating for ARM targets on an x86 host machine.

Usually the front-end of a compiler is considered to be independent of the target architecture. Unfortunately, in our case the front-end of LLVM is partially dependent on host machine and sometimes generates host-specific intermediate code. For example, in case of `long double` data types, the compiler front-end generates operands of type `x86_fp80` in the intermediate code, which are *80-bit* floating point variables specific to the x86 host machines. When this happens, we can neither use the LLVM backend for cross-compilation to another target, say ARM, nor we can annotate the intermediate code for the given target machine and recompile it for the host machine. We have observed this phenomenon in about `86` source files from NewLib only.

Another limitation of the dual compilation flow are the mismatches in the compilation passes executed in the LLVM target and host backends. These mismatches can introduce discrepancies between the control flow graphs of target and host-specific intermediate

representations. This problem appears in the Patricia application and we get considerably inaccurate results. For this reason we removed this application from the final results.

A third source of errors come from the hardware modeling strategy, as KVM has some limitations. Such as the absence of support for for MMX instructions. In general, compiling software using the GCC toolchains does not produce instructions that use MMX registers. But for annotation purposes, the software stack is compiled using the LLVM compilation toolchain, which sometimes generates code that makes explicit use of MMX registers. In order to avoid the generation and use of MMX instructions, we compile and link the applications with software floating-point libraries. These differences introduce further errors in performance evaluation. This partially explains the over estimations in Figure 7.9(b), which means that we execute significantly more instructions for such applications. All of the above measurements are taken for a single processor system, but are equally applicable on multiprocessor systems. Also, these measurements assume that caches are disable in both Rabbits and NaSiK platforms.

Traditionally, native simulators suffer from the lack of correspondence between target and host machine addresses. Our solution improves on this front, as the software is now executed in target address space; that is the data and instruction memory references are very close to the real target platform, although not exactly the same. It means that cache modeling strategies can benefit from this capability, for example when modeling a 32-bit target machine on top of 32-bit host platform, most of the data types and instruction addresses are similar.

In order to remain fast, we can make use of some *heuristics*, either based on simple cache-hit or cache-miss rates, or on much more sophisticated metrics [TAM+08] that take into account the number of load/store instructions in each basic block for estimating the cache effects. Thus, our solution provides a framework for an effective initial design space exploration.

7.4 Multi-processor Experiments

This section provides mostly qualitative results rather than quantitative ones, in order to highlight the feasibility of multiprocessor and hybrid simulation solutions. We emphasize that the proposed solution is scalable and can model from tens to hundreds of processors and yet flexible enough to co-exist within a hybrid multiprocessor simulation platform.

7.4.1 Multi-threaded Applications on SMP Platforms

We use the Parallel-MJPEG application discussed in Section 7.1.2 for testing the multi-threaded application. The application has been configured to create 10 software threads, including 1 dispatcher, 1 serializer and 8 decoder threads. A total of `100` frames are decoded for each execution, and the simulation time is reported using a semi-hosting mechanism. The number of CPUs varies between 1 to 16. Results of the MPSoC simulation are shown in Figure 7.10.

The multi-processor results are a little difficult to comprehend, as the timing behavior of a multiprocessor system depends on many factors. These include timing delays that influence task scheduling, cache contents (if present) and communication sequences on the interconnection network. All of these factors are absent in mono-processor cases, as timing delays in software execution appear exactly in the same order, resulting in consistent system configurations, from one execution to another.

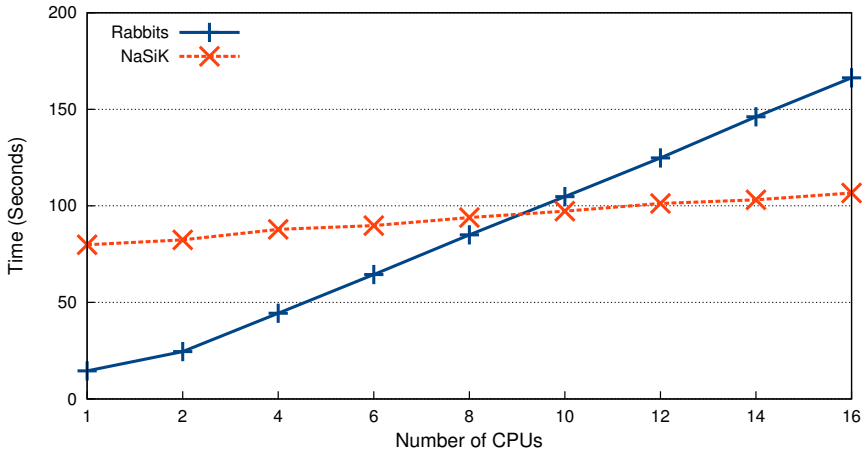


Figure 7.10: MPSoC Simulation Speed Comparison between QEMU and KVM Platforms

CPUs	1	2	4	6	8	10	12	14	16
Rabbits	14.52s	24.52s	44.37s	64.44s	84.98s	104.78s	124.84s	146.15s	166.28s
NaSiK	79.86s	82.36s	87.84s	89.80s	93.98s	97.23s	101.23s	103.13s	106.64s

Table 7.7: Decoding Time for 100 Frames using Parallel-MJPEG Application

These results indicate a key deficiency of the NaSiK solution *i.e.* the costly I/O operations that require guest-to-host mode switching and thus reduce simulation performance. The performance penalty due to I/O is visible for 1 to 8 CPUs where Rabbits performs better than the NaSiK solution. This behavior is application dependent, as we have already discussed it in Section 7.3.2. Another key observation is the scalability of NaSiK platform. As we increase the number of processors, the additional cost for Rabbits solution increases more rapidly as compared to the NaSiK solution. The Rabbits performance penalty is linked to the higher number of SystemC synchronizations, although the simulated time is reduced in MPSoC simulation [Gli10].

7.4.2 Hybrid Simulation Platform

This section details an industrial case study to demonstrate the feasibility of hybrid simulation platforms. We provide an abstract model for such a hybrid platform, namely TI `OMAP L138`, as shown in Figure 7.3. The platform includes two different processors (ARM GPP and a C67 DSP) modeled for MPSoC simulation.

DSPs have a relatively complex instruction sets, which makes it difficult to develop ISSes for them. Moreover, access to DSP compilation tools is often restricted. All native simulation technologies, in general, can solve these issues as there are no ISS development efforts and cross-compilation tools are not necessary. The additional advantage of HAV based approach is to provide the ability to run an unmodified embedded Operating System, the DNA-OS in this case, within the target address space. The ARM is modeled using Rabbits and an unmodified Linux OS runs on it. As presented in Section 5.5, there exists no other native solution in the

literature that profits from such address translation layer and supports hybrid simulation.

In this case study, we present two embedded applications which can benefit from the GPP+DSP structure i.e. a DSP driver which can be part of a more elaborate application, such as a Motion-JPEG decoder and an Audio-Filter application. The best solution is to partition the application and map these partitions to either GPP or DSP depending on their computation or control nature.

The communication between them is also important, in order to build a useful simulation platform. To demonstrate that shared memory based communications are possible, we show a 64-byte data exchange between the two applications in Figure 7.11. Memory accesses are synchronized using a shared memory based global lock, which signals the availability of data to the DSP processor and is located at target address `0xAF000020`. The KVM VCPU models the DSP and starts by waiting for data at the target address `0xAF001000` in step ①. The QEMU processor acts as a DSP driver and starts writing data to the shared memory at step ② and signals the data availability at step ③ by unlocking the shared lock. The KVM processor then starts processing the data at step ④, and returns the filtered data to QEMU using the same mechanism, in reverse direction.

In order to show that such memory hierarchies are possible, we have mapped the shared memory to MMIO based access mechanism for the KVM processor. This option is slower from simulation point of view, as all accesses by the KVM processor cause guest-to-host mode switches. To improve the simulation speed in practical simulation platforms, memory mapping based mechanism should be used (Figure 5.18 Ⓑ and Listing 5.3).

7.5 Simulation of Cross-Compiled DSP Kernels

This section presents some basic results using the SBT technique proposed in Chapter 6. The key idea of the proposed technique is to statically generate native code, with exactly the same *behavior* as it would have on a real VLIW machine, without going through the expensive decoding and pipeline stages. The generated code executes in the extended target address space (Figure 6.4) and accesses to all target specific data locations are made using original addresses. The simulated processor state is also maintained in the extended target address space.

Figure 7.12 shows the simulation performance of compute *vs*. control dominant DSP kernels. These results use three different configurations for the generated code *i.e*. Execute Packet, Hybrid and Inline Hybrid code. Two simulators from *Texas Instruments* (**TI**) are used as a reference *i.e*. *TI-C6x-FCA* and *TI-C6x-DFCA*, as discussed in Section 7.2.3. We use a compute and a control flow dominant application to illustrate the simulation performance trends. The IDCT example crunches on a number of input data blocks and shows the simulation cost of a compute-intensive DSP kernel. The Fibonacci example tests the control dominant behavior, by recursively calling itself twice until an index of 2 or less is reached, resulting in a tree-like structure of function calls. Following three code generation modes are used for the SBT technique:

- SBT (EP): A native function is generated for each VLIW execute packet. Function and Module level optimizations are enabled, while ISA definition inlining is disabled.

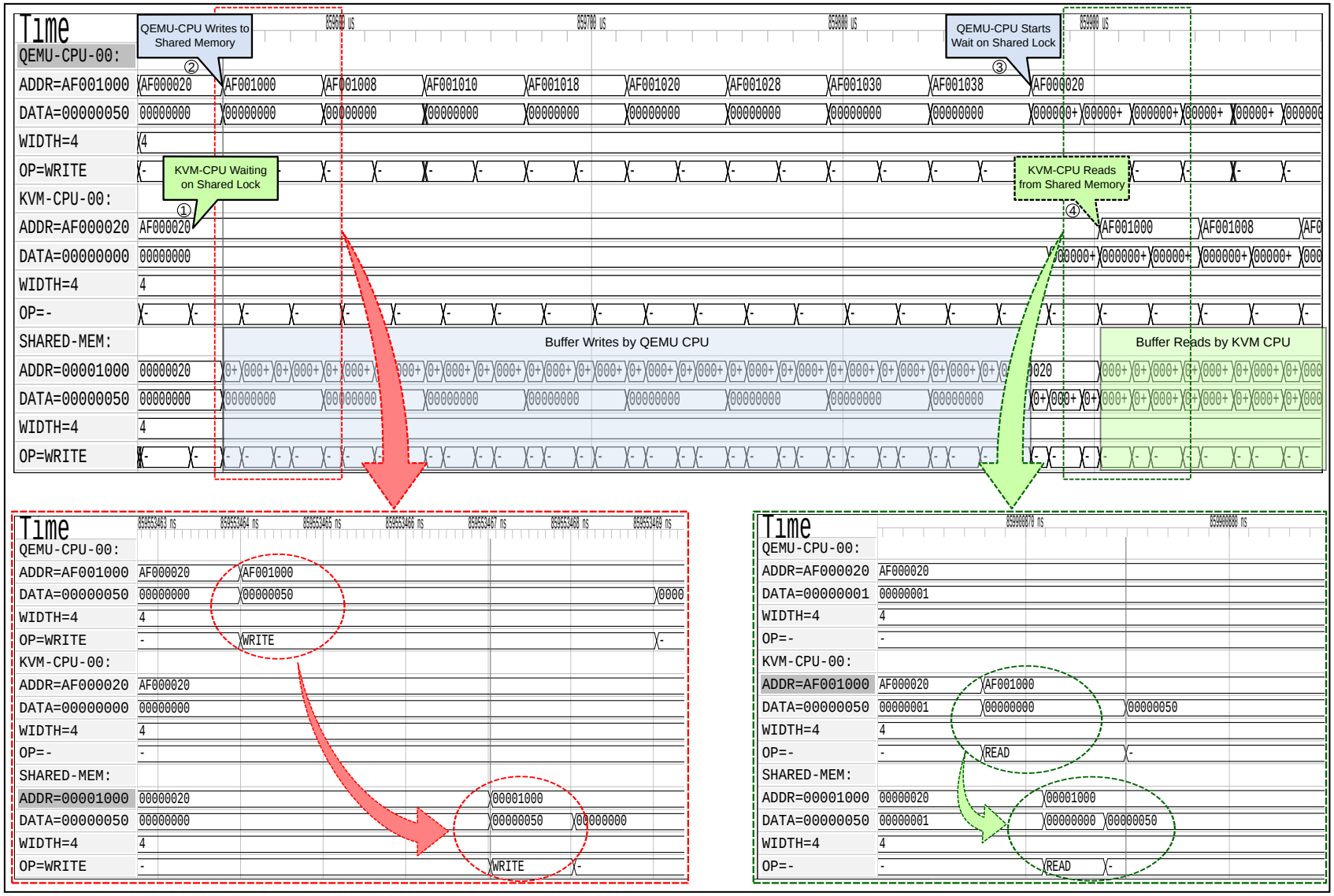
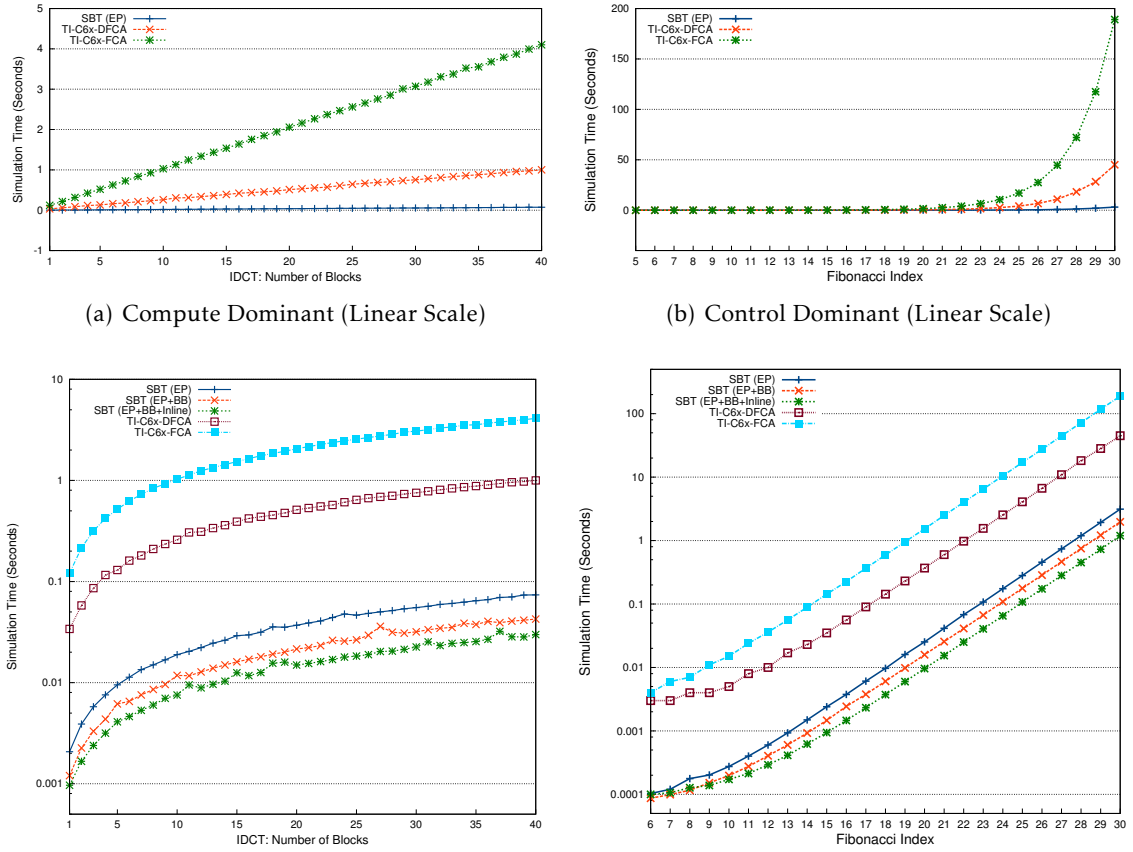


Figure 7.11: Shared Memory Access between QEMU and KVM Processors

- SBT (EP+BB): A native function is generated for each VLIW basic block, as well as for non-startup execute packets. Function and Module level optimizations are enabled with ISA definition inlining disabled.
- SBT (EP+BB+Inline): A native function is generated for each VLIW basic block, as well as for non-startup execute packets. Function and Module level optimizations, and ISA definition inlining are enabled.



(a) Compute Dominant (Linear Scale)

(b) Control Dominant (Linear Scale)

(c) Compute Dominant (Log Scale)

(d) Control Dominant (Log Scale)

Figure 7.12: Performance of Compute *vs*. Control Dominant DSP Kernels

Figures 7.12(a) and 7.12(b) plot results on linear scale, to show the difference in simulation performance between cycle accurate simulators and execute packet level SBT code. We can clearly see that the data processing kernels have a linear trend, which is proportional to the number of data blocks being processed. On the other hand, the control dominant Fibonacci kernel shows an exponential increase in simulation cost, as the amount of control flow increases exponentially with the size of the tree structure being traversed. These are generally well-known trends but the key idea here is to show that cycle accurate simulators cost a lot more in comparison with the natively executing code. Figures 7.12(c) and 7.12(d) plot the same results on logarithmic scale, and add two more results for the static translation case *i.e*. the code generated using the hybrid approach and the code with inlining enabled. These cases are added to show the potential improvements using these code generation modes. A

constant factor improvement between the results obtained from the cycle accurate simulators and the statically generated code is clearly visible.

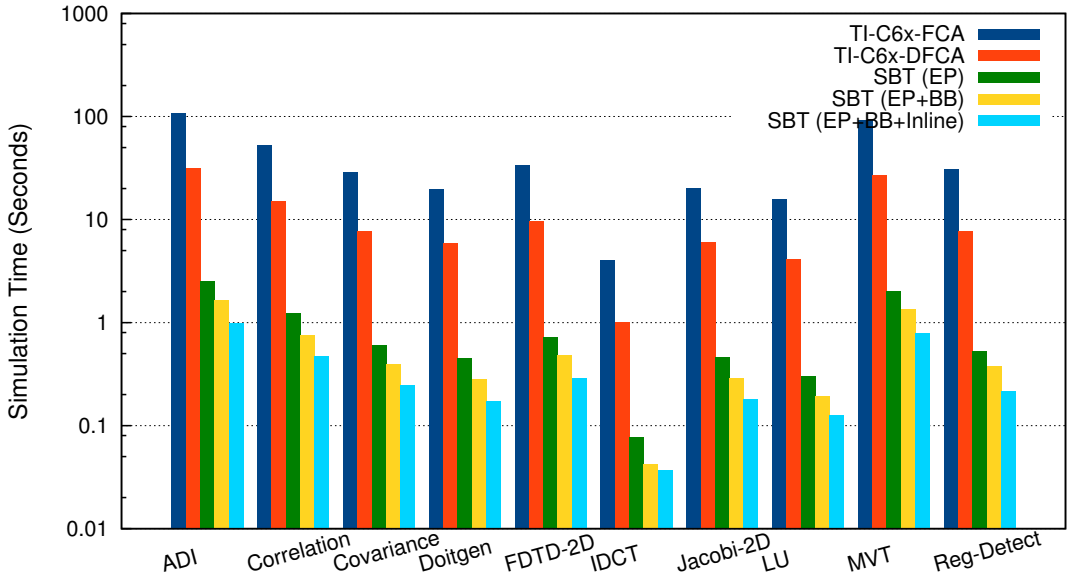


Figure 7.13: Performance Comparison for Different DSP Kernels

To further evaluate the SBT technique, the DSP kernels listed in Section 7.1.4 are used. Out of the three translation options, the execute packet mode is the slowest one, as significant portion of the execution time is spent in control flow between the execute packets, as well as the additional cost for using the non-inlined ISA definitions. To improve the simulation speed, the basic block only mode should be used. However, it is not possible without dynamic translation support, as discussed in Section 6.2.6. Consequently, the hybrid (EP+BB) translation mode is used where all of the non-startup execute packets are translated, in addition to the VLIW basic blocks. Using the hybrid mode the code executes mostly using basic blocks, instead of execute packets. This mode costs more in terms of generated code size, but avoids the dynamic translation requirement as well as improves the simulation speed.

To further improve the simulation speed ISA definitions are inlined in the generated code. Results of these tests are shown in Figure 7.13. We observe a simulation speed-up of more than two orders of magnitude compared to *TI-C6x-FCA* simulator, and an improvement of around 32X compared to *TI-C6x-DFCA* simulator. Table 7.8 compares the results of inlined mode with the cycle accurate simulators and shows a maximum simulation speedup of around 140X and an average speedup of 117X.

The SBT technique is better suited for VLIW software binaries, as we can profit from a higher translation granularity as opposed to the RISC software. Much better results can be achieved, if the VLIW software binaries contain higher level of parallelism in their execute packets, as these are the minimal translation unit for the SBT case. For example, in the TI C6x software, a maximum of eight instructions can be grouped together in parallel. If most of the execute packets contain the maximum number of instructions, the corresponding generated native simulator code will be relatively faster.

The cross-compiled and translated binaries can also profit from the shared memory accesses, as discussed in Section 7.4.2. This feature enables the modeling of heterogeneous

DSP Kernel	TI-C6x-FCA	TI-C6x-DFCA	SBT (EP+BB+Inline)	Speedup *w.r.t.* TI-C6x-FCA	Speedup *w.r.t.* TI-C6x-DFCA
ADI	107.770s	31.718s	0.982s	109.80X	32.31X
Correlation	52.977s	14.970s	0.472s	112.23X	31.71X
Covariance	28.647s	7.718s	0.247s	115.98X	31.25X
Doitgen	19.726s	5.844s	0.172s	114.49X	33.92X
FDTD-2D	33.792s	9.695s	0.290s	116.51X	33.43X
IDCT	4.049s	1.008s	0.037s	109.12X	27.17X
Jacobi-2D	20.201s	6.000s	0.182s	111.18X	33.02X
LU	15.768s	4.144s	0.126s	124.73X	32.78X
MVT	91.198s	26.947s	0.789s	115.63X	34.17X
Reg-Detect	30.657s	7.687s	0.216s	141.99X	35.60X
Average	—	—	—	117.17X	32.54X

Table 7.8: Maximal Speed-ups of SBT Simulation using Hybrid Translation

platforms where different types of software binaries are executed on different types of processing elements to model a heterogeneous MPSoC architecture *e.g.* a GPP+VLIW model.

The proposed SBT technique has a couple of disadvantages. Firstly, as the translation is static, we need to keep a certain level of redundancy in the generated code to avoid runtime translation requirement. Secondly, the translation and optimization time is also a concern. However its a minor issue, as it is performed only once, and it can be efficiently amortized for long running simulations. All in all, simulation times can be improved by using optimized translation toolchains, such as using multiple smaller translation units instead of one large unit, as done in the current implementation. As a last note, the *functional correctness* of generated native simulators has been established using the Execution Trace comparisons with TI simulators.

7.6 Conclusions and Limitations

This chapter provides the necessary experimental evidence to validate our novel approach of implementing native simulation. We use the *Hardware-Assisted Virtualization* (**HAV**) technology as a software interpretation engine within a usual transaction-level MPSoC simulation environment. As opposed to the previous works on this topic, it solves the issue of conflicting and overlapping address spaces, and thus can support complex MPSoC architectures and legacy software. Our experimentations show that the computation and shared memory accesses are not the performance issues for this kind of native MPSoC simulation platforms. Instead, most of the time is spent in I/O accesses, as hardware-assisted virtualization provides a mean to access shared memory transparently, but it must trap for accessing the modeled hardware components. Regardless of this issue, the proposed native simulation technique provides a significant performance gain compared to techniques based on dynamic binary translation.

Support for new ISA is another important aspect of hardware simulation, and we have

two principle approaches for native simulation in such contexts. Firstly, we can add a new target-specific backend to the compilation framework, so we can execute target specific annotation pass and annotate the target independent representation, which is then used to generate native code. Secondly, a static translation of target binary towards native code can be performed and executed in the target address space. We demonstrate the second approach for VLIW ISA and took the example of TI C6x processors. This approach shows performance benefits due to static translation process and HAV based native simulation platform. Certain limitations such code redundancy, can undermine its applicability to general purpose translation contexts. Nevertheless, the SBT approach is useful in early design stages, if precision is more important than simulation speed or modeling of VLIW machines is a requirement.

I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right.

Albert Einstein

8 Conclusions and Perspectives

This thesis presents a novel native simulation technique for MPSoC using the *Hardware-Assisted Virtualization* (**HAV**) technology. The proposed solution uses HAV technology as a software interpretation engine within the usual transaction-level MPSoC simulation environments. As opposed to the previous native techniques, our solution resolves the key issues of conflicting, as well as overlapping address-spaces. Thus, it can support complex MPSoC architectures and legacy software simulation.

Our experiments demonstrate that the computation and shared memory accesses are not the performance issues for this kind of native simulation platforms. Instead, most of time is spent in I/O accesses, as HAV technology provides a mean to access shared memories transparently but must trap for accessing the modeled hardware components. However, the performance gains that our native solution provides compared to the *Dynamic Binary Translation* (**DBT**) based techniques that are currently considered the most efficient ones, is still valuable.

8.1 Conclusions

We presented the key problems of native simulation in Chapter 3, where a set of questions were asked that we intend to answer in this thesis. We repeat the same set of questions here and provide answers to them in the following text.

1. How can we efficiently support the simulated target address-space on host machines without requiring dynamic linking/patching support ?

   In particular, how we can:

   (a) Support the use of SystemC-based hardware IP models, without requiring modifications for address mappings and symbols resolution.

   (b) Minimize software coding constraints, such as the ability to use hard-coded addresses and constant link-time symbols.

(c) Simulate complex operating systems that make use of *Memory Management Unit* (**MMU**) based virtual to physical address translations.

Our principle answer to all of the above questions is based on use of HAV technology to separate the simulated target address-space from the host simulator one. Using the HAV technology, the simulated software gets its own zero-based "physical" address-space in guest mode and these addresses are fixed in nature. The dynamic address allocation from the host user-space is hidden inside the *Virtual Machine Monitor* (**VMM**). Thus, the software is compiled for the target address-space and run-time address translation is provided by the VMM. The memory virtualization feature has two-fold benefit: firstly the hardware models do not need to provide address mappings and symbols and remain un-modified. Secondly the software stack does not require dynamic linking support, as all of the simulated target addresses are known at compile-time.

As the guest-to-host address translation is managed by the VMM, hard-coded addresses can be used, thus, removing the software coding constraints imposed by traditional native techniques. Similarly, the simulated operating system can use its own page tables, which reside in the guest "physical" address-space. All accesses to these structures are either trapped by the VMM to maintain its own set of *Shadow Page Tables* (**SPT**) or hardware based *Extended Page Tables* (**EPT**) to provide guest-to-host memory mapping. Thus, MMU based operating systemes can be used in our native simulation platform.

2. How can we define an automatic software annotation technique into the proposed approach for MPSoC performance estimation ?

   In native techniques, the use of software annotations is considered as the only means for performance estimation. Support for such annotation techniques is possible using the HAV technology, as two different methods for performing I/O operations are supported on x86 machines. We proposed the use of *Memory-Mapped I/O* (**MMIO**) accesses for simulated hardware models and reserving *Port-Mapped I/O* (**PMIO**) for providing a semi-hosting interface. This interface can be used for many purposes, including performance annotations to synchronize with SystemC based hardware models.

3. How can we support native execution of VLIW software, without requiring any runtime support ?

   More specifically, how we can define:

   (a) A VLIW simulation approach, which is generic enough and can also be applied to RISC machines.

   (b) A source-free approach, requiring optimized VLIW binaries only for generating native simulators.

   (c) Accurate support for performance estimation of VLIW software.

   We propose a different solution for VLIW processors, where we use a *Static Binary Translation* (**SBT**) flow to generate native code from cross-compiled target binaries. The proposed solution does not depend on source code availability and is able to generate native code that runs on top of our HAV based simulation platform, without requiring run-time translation support. This solution is generic and well suited to VLIW processors, due to the higher level translation granularity, and at the same time

applicable to RISC machines. The translation flow uses LLVM-IR for two step translation and is retargetable to multiple HAV based simulation platforms. As the translation framework has access to final target instructions, accurate performance annotations can be added to the generated software.

8.2 Perspectives

As the basic technology is now well defined, the future works will focus on finding a solution to minimize the guest-to-host mode switching cost that incurs a high performance penalty during I/O operations. We focused on reducing the number of such transitions, whereas future works should consider the cost of individual transitions and how it could be reduced.

The use of IOMMU is possible in virtual machines for direct access to peripheral devices but in case of simulation models such I/O accesses must go through SystemC communication components. Another future direction would be to see if these two conflicting requirements could be satisfied and remove the I/O bottleneck from HAV based simulation.

The annotation technique used in this thesis is useful only for simple RISC machines and provides reasonable results. This is one of the reasons why we proposed the SBT technique for VLIW processors. Future works in annotation context should focus on more complex architectures such as superscalar and VLIW machines and see if the same dual compilation principle can be extended for these processors.

For the SBT technique, the ISA definition and instruction decoder generation process should be automated using a generic processor description language.

A

Sensitive and Unprivileged Instructions in IA-32 (x86) Architectures

This appendix briefly describes the sensitive and unprivileged instruction found in IA-32 (x86) architectures. A sensitive instruction can consult and modify the processor state, thus effecting the execution mode of operating system and/or the *Virtual Machine Monitor* (**VMM**). Table A.1 shows the possible scenarios for software based virtualization techniques, with the problematic case when sensitive and unprivileged instructions are present in the *Instruction Set Architecture* (**ISA**) under question. In order for an architecture to be efficiently virtualizable, sensitive instructions from the guest operating system must not directly execute on the host processor but must be trapped and emulated by the VMM. Table A.2 and Table A.3 detail sensitive but unprivileged instructions found in x86 architectures [RI00], thus Popek and Goldberg [PG74] virtualization requirements cannot be met in the presence of these instructions, without additional hardware support.

	Sensitive	**Non-Sensitive**
Privileged	Trap and emulate	Trap but no need to emulate (Rare case)
Unprivileged	*Cannot trap but must emulate*	Direct execution

Table A.1: Sensitive/Non-Sensitive *vs*. Privileged/Unprivileged Instructions

Most of the sensitive and unprivileged instructions described in Table A.2 and Table A.3 exhibit a single type of problem, usually emerging from the privilege level of guest software. A VMM must preserve the illusion that the guest software executes at its intended privilege level, thus it must emulate all of the sensitive instructions whether privileged or unprivileged. Following list details the problems associated with the unprivileged execution of sensitive instructions that prevent efficient virtualization of x86 architectures.

Instruction	Description	Problem
PUSHF (16-bit) PUSHFD (32-bit)	Pushes the lower 16 (32) bits of the EFLAGS register onto the stack and decrements the stack pointer by 2 (4).	Ⓐ
POPF (16-bit) POPFD (32-bit)	Pops a word from the top of the stack, increments the stack pointer by 2 (4), and stores the value in the lower 16 (32) bits of the EFLAGS register.	Ⓐ
SGDT	Stores the contents of the GDTR in a 6-byte memory location.	Ⓑ
SIDT	Stores the contents of the IDTR in a 6-byte memory location.	Ⓑ
SLDT	Stores the segment selector from the LDTR in a 16 or 32-bit general-purpose register or memory location.	Ⓑ
SMSW	Stores the machine status word (bits 0 through 15 of CR0) into a general-purpose register or a memory location.	Ⓑ

Table A.2: Sensitive Register Instructions in IA-32 (x86)

Ⓐ This type of problem appears in instructions that are sensitive and unprivileged *i.e.* they do not generate traps when executed with insufficient privileges, but their effect may not take place. Moreover, if successfully executed, these instructions can allow a guest operating system to see and/or modify the processor state; thus, deducing the existence of a *Virtual Machine* (**VM**) and effecting itself or the operating mode of VMM.

Ⓑ This type of problem appears in instructions that are sensitive and unprivileged as well. Such instructions allow the guest operating system to see the values of system registers and segments *e.g.* GDT, LDT, IDT, CR0, CS, SS *etc.*, even though the guest operating system executes in the unprivileged mode. A guest might find unexpected values in system registers/segments and behave incorrectly or halt completely.

Ⓒ These instructions include references to protection system and include checks to see if the Current Privilege Level (CPL) and/or Requested Privilege Level (RPL) are greater than the Descriptor Privilege Level (DPL). A guest operating system usually executes in unprivileged mode, but assumes that it is privileged; thus, it can access any segment descriptors with its user mode CPL and cause problems. Some of the instructions exhibiting this type of problem may try to access call gates or task gates of higher privilege level as well *e.g.* as in CALL and JMP instructions.

Instruction	Description	Problem
CALL	Saves procedure linking information to the stack and branches to the procedure given in its destination operand.	Ⓒ
INT *n*	Performs a call to the interrupt or exception handler specified by *n*. It also pushes the EFLAGS register onto the stack in addition to the return address.	Ⓒ
JMP	Transfers program control to another location in the instruction stream but does not record the return address.	Ⓒ
LAR	Loads access rights from a segment descriptor into a general-purpose register.	Ⓒ
LSL	Loads the segment limit from a segment descriptor into a general-purpose register.	Ⓒ
MOV	Copies the source operand value to the destination operand. The source operand can be an immediate value, general-purpose register, segment register or a memory location; the destination operand can be a general-purpose register, segment register or a memory location.	Ⓒ
POP	Loads a value from the top of the stack to a general-purpose register, memory location or a segment register.	Ⓒ
PUSH	Allows a general-purpose register, memory location, an immediate value or a segment register to be pushed onto the stack.	Ⓑ
RET	Transfers program control to a return address that is placed on the stack (normally by a CALL instruction).	Ⓐ Ⓒ
STR	Stores the segment selector from the task register into a general-purpose register or a memory location.	Ⓐ Ⓒ
VERR	Verifies whether a code or data segment is readable from the Current Privilege Level (CPL).	Ⓒ
VERW	Verifies whether a code or data segment is writable from the Current Privilege Level (CPL).	Ⓒ

Table A.3: Sensitive Protection System Instructions in IA-32 (x86)

B

Memory Virtualization Support in Hardware-Assisted Virtualization

Memory Management Unit (**MMU**) based Operating Systems maintain a mapping between virtual and physical address spaces using page tables. Each time the CPU generates a virtual address, the *Virtual Page Number* (**VPN**) of the address is looked up in the *Translation Lookaside Buffer* (**TLB**), which is an associative address translation cache. In case of a TLB hit, the physical address is calculated from the *Physical Frame Number* (**PFN**) retrieved from the TLB and the offset part of the virtual address. For a TLB miss, the paging structures are used to translate the VPN to its corresponding PFN. The TLB is updated as well for later accesses to the same VPN. The key feature of this translation mechanism is the use of TLB, as TLB hit rate is usually very high and page table walks are quite rare. This processes is illustrated in Figure B.1.

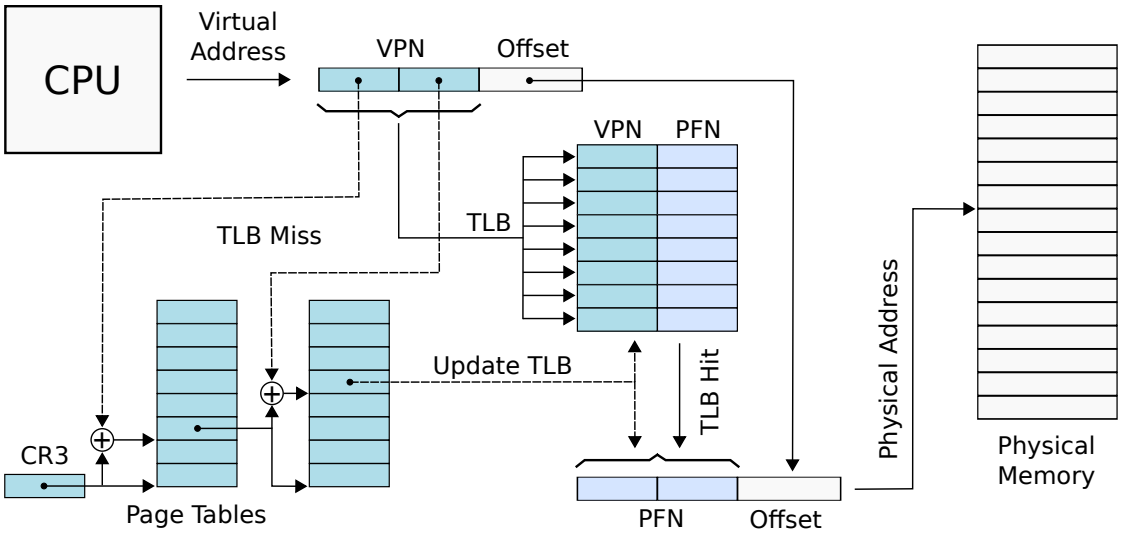


Figure B.1: Virtual to Physical Address Translation and Translation Lookaside Buffer

B.1 Memory Virtualization using Shadow Page Tables

In a virtualized environment, a guest operating system expects access to the "physical" memory of the system, once it has obtained a physical address through its paging structures. The guest OS maintains its own set of page tables and uses MMU for performing *Guest Virtual Address* (**GVA**) to *Guest Physical Address* (**GPA**) translations. In order to access the real physical memory of the system, an additional translation step is necessary to convert the *Guest Physical Addresses* (**GPAs**) to *Host Physical Addresses* (**HPAs**). This is a costly operation, as every address translation has to go through two different paging structures. *Shadow Page Tables* (**SPT**) avoid this double bookkeeping by skipping the intermediate translation from GVAs to GPAs. Each *Shadow Page Table Entry* (**SPTE**) makes a direct translation from GVA to HPA and provides the MMU and TLB with these translations, when operating in guest mode. Figure B.2 illustrates the role of SPTs in guest address translation.

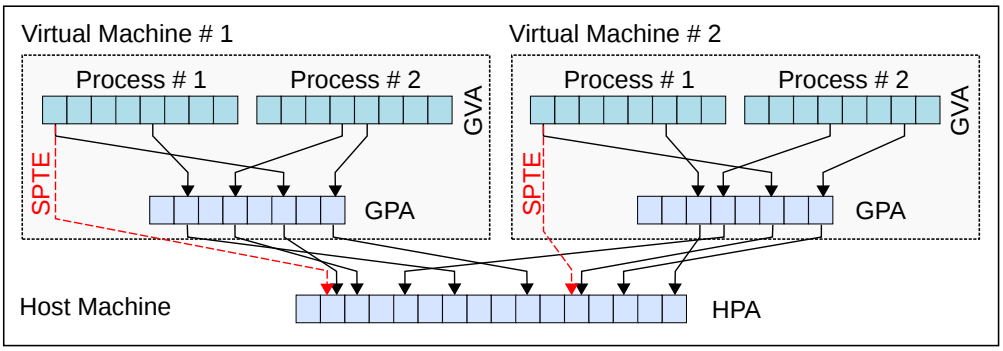


Figure B.2: Role of Shadow Page Tables in Guest Address Space Translation [VMW09]

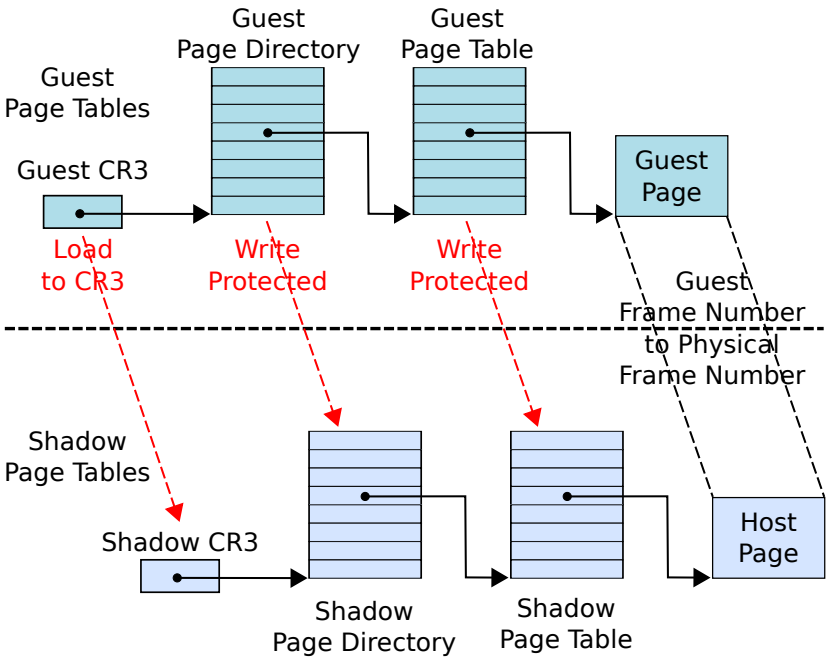


Figure B.3: Guest Page Table Accesses and Synchronization with Shadow Page Tables

SPTs cause performance penalty as each update of the guest page tables requires an equivalent bookkeeping on the shadow pages. The VMM achieves this objective by write protecting the guest paging structures, as well as intercepting guest paging operations including page faults, page invalidations and loads to the guest CR3 register. For example, when load to the guest CR3 register is trapped by VMM, the hardware TLB is flushed and new shadow page directory entry is loaded in the shadow CR3 register. This process is shown in Figure B.3.

The SPT synchronization operations cause significant overheads in case of HAV based virtual machines, as the costly VM Exit and VM Entry operations are required for such synchronizations. VMM keeps the GPA to HPA mappings in its internal data structures and exposes the GVA to HPA mappings to the actual hardware. The most recently used GVA to HPA translations are cached in the hardware TLB and are used to transparently access the host machine memory, from the guest.

B.2 Memory Virtualization using Extended Page Tables

In *Extended Page Tables* (**EPT**) (also known as *Rapid Virtualization Indexing* (**RVI**)) based systems, the guest maintains its own paging structures and performs GVA to GPA translations, similar to a non-virtualized system. The VMM maintains an additional level of page tables, known as Extended or Nested page tables, which provide GPA to HPA translations, as shown in Figure B.4.

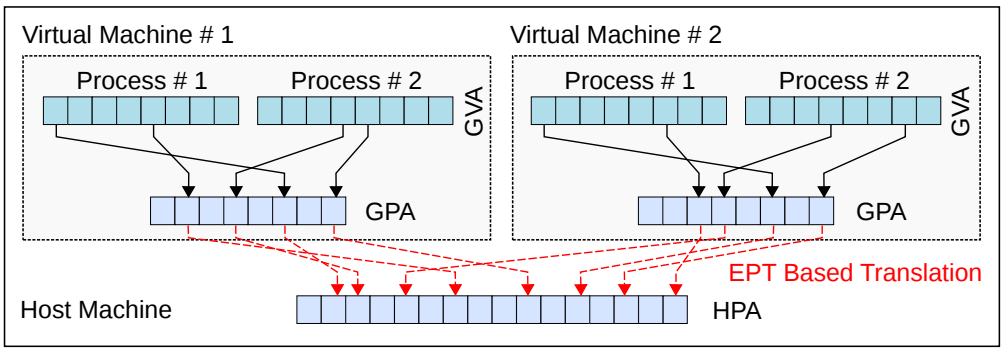


Figure B.4: Role of Extended Page Tables in Guest Address Space Translation [VMW09]

In EPT, both the guest and nested page tables are available to the address translation hardware. For each TLB miss, the hardware page table walker traverses guest paging structures and for each GPA access during this process, it also walks the nested paging structures to get the corresponding HPA. This additional translation step removes the need for shadow page tables and their synchronization with the guest paging structures. This process is illustrated in Figure B.5.

The translation process now becomes costly, as a TLB miss can cause many references to the guest and nested page tables. This additional cost can adversely effect the performance of applications with high TLB miss rates, as compared to the shadow page tables. In cases when the guests frequently update their paging structures, the performance of EPT is better than the shadow pages, as guest-to-host mode switches are not necessary. Moreover, the translation process does not require additional memory for shadow paging structures for the guest processes.

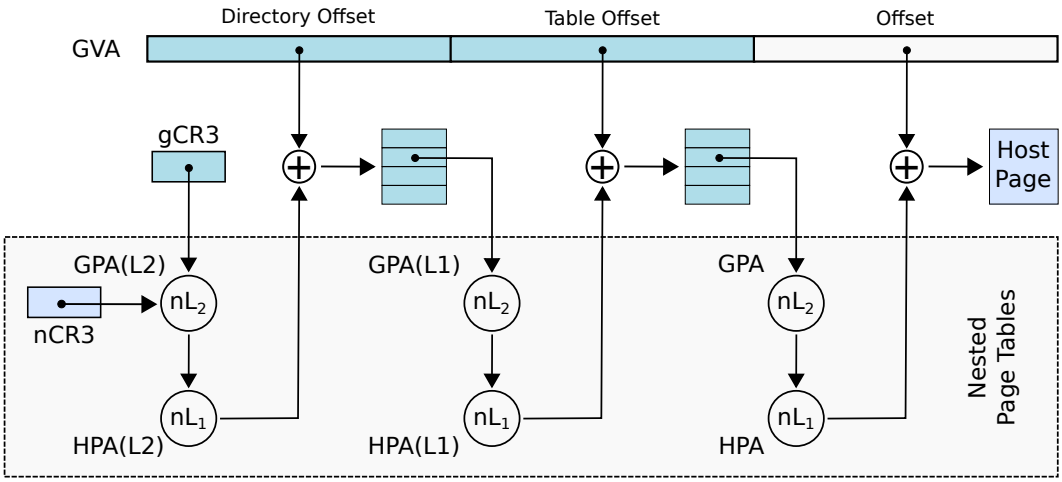


Figure B.5: Guest Physical Address Translation using Extended Page Tables

Caching techniques for page table walks and TLBs are employed to reduce the cost of EPT based address translation. The page walk cache mechanism provides hardware based support for saving intermediate translations, which are accessed first to see if an intermediate translation is available before going through the nested paging structures. Similarly, nested TLBs are used to keep the most recently accessed GPA to HPA mappings in hardware, to accelerate the EPT translation process.

C

Code Generation Algorithms for VLIW Software Simulation

We use a set of algorithms in our *Static Binary Translation* (**SBT**) technique and generate LLVM-IR code from cross-compiled VLIW binaries, discussed in Chapter 6. The generated code is compiled for the host machine and executed on top of our *Hardware-Assisted Virtualization* (**HAV**) based native simulation platform, as discussed in Chapter 5. Following sections provide details on these algorithms and explain the IR code generation process, as discussed in Section 6.2.3 and Figure 6.2.

C.1 IR Generation for VLIW Basic Blocks

Algorithm 1 is the entry point in code generation process, which takes a VLIW basic block as input and produces a corresponding function in LLVM-IR. It creates the overall structure of the IR function *i.e.* the *Entry*, *Return*, *Core* and *Update* basic blocks and the control flows between them. It does not create instructions that simulate the VLIW instructions, for this purpose it uses the Algorithm 2 given in the following section.

The algorithm starts by creating Entry and Return IR basic blocks at Lines 3 and 4, respectively. The main loop at Line 6 repeats for each execute packet in the VLIW basic block. A pair of Core and Update IR basic blocks is created at Lines 7 and 8, followed by a check to see if the previous Update basic block exists *i.e.* $BB_{Last} \neq \phi$. If true, comparison and conditional branch instructions are created at Lines 10 and 11 and pushed at the end of previous Update basic block.

The Algorithm 2 is invoked at Line 13 to generate code for the current execute packet. Finally the algorithm links Entry basic block to the first Core basic block at Line 17 and the last Update basic block to the Return basic block, at Line 18, creating the overall structure of an IR function.

Algorithm 1 Code Generation for Each VLIW Basic Block

Input: A VLIW Basic Block.
Output: A Function in LLVM-IR.

1: **procedure** $GenerateIR_{BB}(BB_{VLIW})$
2: Set $RPC_T = \phi$ and $BB_{Last} = \phi$
3: $BB_{Entry} \leftarrow$ Create $BasicBlock_{IR}$
4: $BB_{Return} \leftarrow$ Create $BasicBlock_{IR}$
5: $I \leftarrow 1$
6: **for each** EP_{VLIW} in BB_{VLIW} **do**
7: $BB_{Core}[I] \leftarrow$ Create $BasicBlock_{IR}$
8: $BB_{Update}[I] \leftarrow$ Create $BasicBlock_{IR}$
9: **if** $BB_{Last} \neq \phi$ **then**
10: $Inst_{CMP} \leftarrow$ Create a comparison instruction for $RPC_T \neq \phi$
11: Create a Branch to BB_{Return} if $Inst_{CMP}$ is set else to $BB_{Core}[I]$
12: **end if**
13: $RPC_T \leftarrow$ CALL $GenerateIR_{EP}(EP_{VLIW}, BB_{Core}[I], BB_{Update}[I], BB_{Return})$
14: $BB_{Last} \leftarrow BB_{Update}[I]$
15: $I \leftarrow I + 1$
16: **end for**
17: Create Branch Instruction to $BB_{Core}[1]$ in BB_{Entry}
18: Create Branch Instruction to BB_{Return} in BB_{Last}
19: **return**
20: **end procedure**

C.2 IR Generation for VLIW Execute Packets

Algorithm 2 operates at Core and Update IR basic block pairs, for each execute packet. It generates memory allocation instruction for saving results of ISA executions at Line 2 (Figure 6.2 (C1)). $Inst_{ET}$ is the IR function level early termination condition and $Inst_{RISA}$ is used to temporarily store the return value of each ISA execution.

The main loop at Line 5, executes for each VLIW instruction in the current execute packet. It gets the decoded VLIW instruction and asks it to generate an ISA call for itself at Lines 6 and 7, respectively (Figure 6.2 (C2)). For ISA call generation, it provides the corresponding $Inst_{Res}[]$ memory reference as input, along with the generic parameters including simulated processor state. The generated ISA behavior call serves as a reference and will be optimized-out at later.

Algorithm 3 is invoked at Line 11, to insert Immediate or Buffered update calls in the current Core basic block (Figure 6.2 (C3)). Instructions at Lines 12 and 13 load the function level early termination value and create a branch to Return or the Update basic block, depending on the value of $Inst_{ET_C}$ flag. Finally, this algorithm creates the simulated processor state update calls at Lines 14 and 15 (Figure 6.2 (U1)). The `UpdateRegs` function performs the delayed update of registers, including the target program counter.

Algorithm 3 inserts post update calls, either `ImmediateUpdate` or `BufferedUpdate` for ISA results, in the current Core basic block (Figure 6.2 (C3)). It iterates over all instructions in the given execute packet, gets the decoded target instruction $Inst_{DVLIW}$ to determine the number of delay slots, as shown at Lines 4 and 5. Depending upon the number of delay slots, it either adds a call to `BufferedUpdate` at Line 8 or `ImmediateUpdate` at Line 14. This algorithm also handles side-effects by inserting an `ImmediateUpdate` call at Line 11, as side effects in C6x processors have *zero* delay slots.

Algorithm 2 Code Generation for Each VLIW Execute Packet

Input: A VLIW Execute Packet along with Core, Update and Return Basic Blocks in LLVM-IR.
Output: LLVM-IR Code for VLIW Execute Packet, Processor State Updates and Control Flows between the Input IR Basic Blocks.

function $GenerateIR_{EP}(EP_{VLIW}, BB_{Core}, BB_{Update}, BB_{Return})$
  Allocate Stack Space for $Inst_{Res}[]$ of $EP_{VLIW}.Size()$
  Set $Inst_{ET} = \phi$ and $Inst_{RISA} = \phi$
  $J \leftarrow 1$
  for each $Inst_{VLIW}$ in EP_{VLIW} **do**
    $Inst_{DVLIW} \leftarrow$ Get Decoded Instruction for $Inst_{VLIW}$
    $Inst_{RISA} \leftarrow$ Ask $Inst_{DVLIW}$ to place an ISA call in the current IR module using its Operand Types, Operands and $Inst_{Res}[J]$ for saving ISA results.
    Update early termination condition $Inst_{ET}$ using $Inst_{RISA}$.
    $J \leftarrow J + 1$
  end for
  CALL $GenerateIR_{PostUpdates}(EP_{VLIW}, BB_{Core}, Inst_{Res}[])$
  Create instruction for testing the early termination $Inst_{ET_C} \leftarrow Inst_{ET} \neq 0$
  Create a Branch to BB_{Return} if $Inst_{ET_C}$ is set else to BB_{Update}
  Create Calls for $IncPC()$ and $IncCycles()$
  Create Call for $UpdateRegs()$ and save its return value in $Inst_{RPC_T}$
  return $(Inst_{RPC_T})$
end function

Algorithm 3 Immediate or Buffered Register Updates

Input: A VLIW Execute Packet along with Core IR Basic Block and Results of VLIW ISA Executions.
Output: LLVM-IR Code for Buffered or Immediate Updates and handling Side Effects.

procedure $GenerateIR_{PostUpdates}(EP_{VLIW}, BB_{Core}, Inst_{Res}[])$
  $K \leftarrow 1$
  for each $Inst_{VLIW}$ in EP_{VLIW} **do**
    $Inst_{DVLIW} \leftarrow$ Get Decoded Instruction for $Inst_{VLIW}$
    $DelaySlots \leftarrow$ Get Delay Slots from $Inst_{DVLIW}$
    if $DelaySlots \neq 0$ **then**
      if $Inst_{DVLIW}.Type \neq STORE$ **then**
        Create Call to $BufferedUpdate(Inst_{Res}[K], DelaySlots)$
      end if
      if $Inst_{DVLIW}.Type = LOAD \parallel Inst_{DVLIW}.Type = STORE$ **then**
        Create Call to $ImmediateUpdate(Inst_{Res}[K])$
      end if
    else
      Create Call to $ImmediateUpdate(Inst_{Res}[K])$
    end if
    $K \leftarrow K + 1$
  end for
  return
end procedure

Glossary

ABI	Application Binary Interface
ADC	Analog to Digital Converter
AMP	Asymmetric Multi-Processor
APES	APplication Elements for System-on-chips
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
ARM	Advanced RISC Machines
CA	Cycle Accurate
CAD	Computer-Aided Design
CFG	Control Flow Graph
CMP	Chip Multiprocessor
CPU	Central Processing Unit
DBT	Dynamic Binary Translation
DES	Discrete Events Simulation
DMA	Direct Memory Access
DMI	Direct Memory Interface
DSE	Design Space Exploration
DSP	Digital Signal Processor
EPT	Extended Page Tables
EU	Execution Unit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GDB	GNU DeBugger
GDTR	Global Descriptor Table Register
GPA	Guest Physical Address
GPP	General Purpose Processor
GPOS	General Purpose Operating System
GVA	Guest Virtual Address
HAL	Hardware Abstraction Layer
HAV	Hardware-Assisted Virtualization
HDL	Hardware Description Language
HDS	Hardware Dependent Software
HIS-CS	Hybrid Instruction-Set-Compiled Simulation
HPA	Host Physical Address
HRTL	Higher-Level Register Transfer Language
HVA	Host Virtual Address
HVM	Hardware Virtual Machine
IOAPIC	I/O APIC
IDT	Interrupt Descriptor Table
IDTR	Interrupt Descriptor Table Register
IOMMU	Input/Output Memory Management Unit
IP	Intellectual Property
IDCT	Inverse Discrete Cosine Transform
ILP	Instruction Level Parallelism
IPC	Inter Process Communication

IR	Intermediate Representation
ISA	Instruction Set Architecture
IS-CS	Instruction Set Compiled Simulation
ISS	Instruction Set Simulator
IPI	Inter-Processor Interrupt
ITRS	International Technology Roadmap for Semiconductors
JIT-CCS	Just-In-Time Cache-Compiled Simulation
JPEG	Joint Photographic Experts Group
KVM	Kernel Virtual Machine
LAPIC	Local APIC
LDTR	Local Descriptor Table Register
LLVM	Low Level Virtual Machine
MJPEG	Motion-JPEG
MMIO	Memory-Mapped I/O
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
MMX	MultiMedia eXtension
MPSoC	Multi-Processor System-on-Chip
MSR	Model Specific Register
NGPA	Nested Guest Physical Address
NGVA	Nested Guest Virtual Address
NOP	No Operation
NPU	Native Processing Unit
OS	Operating System
PFN	Physical Frame Number
PIC	Programmable Interrupt Controller
PIT	Programmable Interval Timer
PMIO	Port-Mapped I/O
POSIX	Portable Operating System Interface
PV	Programmer's View
PVT	Programmer's View with Time
PW	Processor Wrapper
QEMU	Quick EMUlator
RAM	Random Access Memory
RAW	Read After Write
RISC	Reduced Instruction Set Computers
RPC	Remote Procedure Call
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RVI	Rapid Virtualization Indexing
SBT	Static Binary Translation
SIMD	Single Instruction, Multiple Data
SL	System Level
SMC	Self-Modifying Code
SMP	Symmetric Multi-Processor
SoC	System-on-Chip
SPT	Shadow Page Tables
SPTE	Shadow Page Table Entry
TA	Transaction Accurate
TI	Texas Instruments
TLB	Translation Lookaside Buffer
TLM	Transaction Level Modeling
TLM-T	Transaction Level Modeling with Time
TLM-DT	Transaction Level Modeling with Distributed Time
TR	Task Register
VA	Virtual Architecture

VCPU Virtual CPU

VHDL VHSIC Hardware Description Language

VLIW Very Long Instruction Word

VLSI Very-Large-Scale Integration

VM Virtual Machine

VMM Virtual Machine Monitor

VMCB Virtual Machine Control Block

VMCS Virtual Machine Control Structure

VMX Virtual Machine eXtensions

VPID Virtual Processor Identifier

VPN Virtual Page Number

WAR Write After Read

WAW Write After Write

WCET Worst Case Execution Time

XML Extensible Markup Language

List of Publications

International Conferences

[1] **Mian Muhammad Hamayun** Frédéric Pétrot and Nicolas Fournel. Native Simulation of Complex VLIW Instruction Sets using Static Binary Translation and Hardware-Assisted Virtualization. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan*, pages 576–581, January 2013.

[2] Patrice Gerin, **Mian Muhammad Hamayun**, and Frédéric Pétrot. Native MPSoC Co-Simulation Environment for Software Performance Estimation. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2009, Grenoble, France*, pages 403–412, 2009.

International Journals

[3] Hao Shen, **Mian Muhammad Hamayun**, and Frédéric Pétrot. Native Simulation of MPSoC Using Hardware Assisted Virtualization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):1074–1087, July 2012.

[4] Frédéric Pétrot, Nicolas Fournel, Patrice Gerin, Marius Gligor, **Mian Muhammad Hamayun**, and Hao Shen. On MPSoC Software Execution at the Transaction Level. *IEEE Design & Test of Computers*, 28(3):2–11, 2010.

Book Chapters

[5] Frédéric Pétrot, Patrice Gerin, and **Mian Muhammad Hamayun**. On Software Simulation for MPSoC: A Modeling Approach for Functional Validation and Performance Estimation. In Gabriela Nicolescu, Ian O'Connor, and Christian Piguet, editors, *Design Technology for Heterogeneous Embedded Systems*, pages 91–114. Springer, 2012.

References

[Acc] Accellera Systems Initiative, http://www.accellera.org. 3.3.1

[AMD05] *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005. Publication No. 33047, Revision 3.01. 5

[BBCR02] I. Barbieri, M. Bariani, A. Cabitto, and M. Raggio. Multimedia-application-driven Instruction Set Architecture Simulation. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, volume 2, pages 169 – 172, 2002. 4.2.3

[BBY+05] Aimen Bouchhima, Iuliana Bacivarov, Wassim Youssef, Marius Bonaciu, and Ahmed A. Jerraya. Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration. In *Proceedings of the 10th Asia South Pacific Design Automation Conference*, pages 969–972, 2005. 1.1, 1.2.1, 2.1, 3.4.4.2, 3.5, 3.5.2, 4.1.2

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. 5

[Bel73] James R. Bell. Threaded Code. *Communication of the ACM*, 16(6):370–372, 1973. 1.1, 2.1

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. 1.1, 2.1, 4.2.3.2, 5.5, 6.1, 6.2.1

[Ber06] V. Berman. Standards: The P1685 IP-XACT IP Metadata Standard. *IEEE Design & Test of Computers*, 23(4):316–317, April 2006. 3.4

[BG04] M. Burtscher and I. Ganusov. Automatic Synthesis of High-Speed Processor Simulators. In *37th International Symposium on Microarchitecture*, pages 55 – 66, December 2004. 4.2.3.1

[BHK+00] J.R. Bammi, E. Harcourt, W. Kruitzer, L. Lavagno, and M.T. Lazarescu. Software Performance Estimation Strategies in a System-Level Design Tool. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 82 –86, May 2000. 4.2.1

[BK07] Milos Becvár and Stanislav Kahánek. VLIW-DLX Simulator for Educational Purposes. In Edward F. Gehringer, editor, *WCAE*, pages 8–13, 2007. 4.2.3

[BNH+04] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1625–1639, December 2004. 4.2.3.2

[BPN+04] Alex Bobrek, Joshua J. Pieper, Jeffrey E. Nelson, JoAnn M. Paul, and Donald E. Thomas. Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 1144–1149, Washington, DC, USA, 2004. 4.1.3

[BYJ02] M. Bacivarov, Sungjoo Yoo, and A.A. Jerraya. Timed HW-SW Cosimulation using Native Execution of OS and Application SW. In *Seventh IEEE International High-Level Design Validation and Test Workshop*, pages 51 – 56, October 2002. 1.2, 4.1.1

[BYJ04] Aimen Bouchhima, Sungjoo Yoo, and Ahmed Jeraya. Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 469–474, Piscataway, NJ, USA, 2004. 1.2.1, 1.2.1, 4.1.2, 4.1.2

[CBR+04] Jerome Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois, El Mostapha Aboulhamid, and Francois-Raymond Boyer. SPACE: A Hardware/Software SystemC Modeling Platform Including an RTOS. In *Languages for System Specification*, pages 91–104. 2004. 4.1.1

[CG03] Lukai Cai and Daniel Gajski. Transaction Level Modeling: An Overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 19–24, New York, NY, USA, 2003. 3.4.2, 4.1.2

[CGG04] Lukai Cai, A. Gerstlauer, and D. Gajski. Retargetable Profiling for Rapid, Early System-level Design Space Exploration. In *Proceedings of the 41st ACM/IEEE Design Automation Conference*, pages 281–286, July 2004. 3.5.5, 4.2

[CHB09a] E. Cheung, H. Hsieh, and F. Balarin. Memory Subsystem Simulation in Software TLM/T Models. In *Proceedings of the 14th Asia South Pacific Design Automation Conference, Yokohama, Japan*, pages 811–816, January 2009. 1.2, 4.1.1

[CHB09b] Eric Cheung, Harry Hsieh, and Felice Balarin. Fast and Accurate Performance Simulation of Embedded Software for MPSoC. In *Proceedings of the 14th Asia South Pacific Design Automation Conference, Yokohama, Japan*, pages 552–557, January 2009. 3.5.5, 4.1.1, 4.2.2

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32 - A Profile-Directed Binary Translator. *IEEE Micro*, 18:56–64, 1998. 4.2.3.1, 4.2.3.2

[CK94] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994. 4.2.3.2

[CMMC08] J. Cornet, F. Maraninchi, and L. Maillet-Contoz. A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 9–14, March 2008. 3.4.4.1

[Cre81] Robert Jay Creasy. The Origin of the VM/370 Time-sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981. 1.3, 5

[CSC+09] Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A High-level Virtual Platform for Early MPSoC Software Development. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, Grenoble, France*, pages 11–20, 2009. 1.2.1, 4.1.2

[CVE00] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *IEEE Computer*, 33(3):60–66, March 2000. 4.8, 4.2.3.1, 6.1, 6.2.5

[CYH+08] Jiunn-Yeu Chen, Wuu Yang, Tzu-Han Hung, Charlie Su, and Wei Chung Hsu. On Static Binary Translation and Optimization for ARM based Applications. In *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*, April 2008. 4.2.3.1, 6.1, 6.2.5

[DGB+03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Er Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 15–24, 2003. 4.2.3.2

[DGM09] Rainer Dömer, Andreas Gerstlauer, and Wolfgang Müller. Introduction to Hardware-dependent Software Design Hardware-dependent Software for multi- and many-core Embedded Systems. In *Proceedings of the 14th Asia South Pacific Design Automation Conference, Yokohama, Japan*, pages 290–292, Piscataway, NJ, USA, 2009. 3.2

[DM99] G. De Micheli. Hardware Synthesis from C/C++ Models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 1999*, pages 382–383, 1999. 3.3

[Don04] Adam Donlin. Transaction Level Modeling: Flows and Use Models. In *Proceedings of the 2nd IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 75–80, New York, NY, USA, 2004. 3.4.2

[EA97] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. *SIGARCH Computer Architecture News*, 25(2):26–37, May 1997. 4.2.3.2

[EHV09] Wolfgang Ecker, Stefan Heinen, and Michael Velten. Using a Dataflow Abstracted Virtual Prototype for HdS-design. In *Proceedings of the 14th Asia South Pacific Design Automation Conference, Yokohama, Japan*, pages 293–300, 2009. 4.1.1, 4.2.2

[Fis83] Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 11:3:140–150, June 1983. 4

[Fis09] J.A. Fisher. Retrospective: Very Long Instruction Word Architectures and the ELI-512. *IEEE Solid-State Circuits Magazine*, 1(2):34 –36, spring 2009. 4

[GCDM92] R. K. Gupta, C. N. Coelho, Jr., and G. De Micheli. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 225–230, 1992. 1.2, 4.1.1

[GCKR12] A. Gerstlauer, S. Chakravarty, M. Kathuria, and P. Razaghi. Abstract System-Level Models for Early Performance and Power Exploration. In *Proceeding of the 17th Asia and South Pacific Design Automation Conference*, pages 213–218, February 2012. 1.1, 2.1, 4.1.1

[Ger09] Patrice Gerin. *Modéles de Simulation pour la Validation Logicielle et l'Exploration d'Architectures des Systémes Multiprocesseurs sur Puce.* Ph.D Thesis, Institut Polytechnique de Grenoble, Grenoble, November 2009. (document), 1.7, 1.4, 3.5, 3.5.2, 3.6, 3.7, 3.4, 3.5, 3.5.4, 4.1.2.2, 5.2.3, 5.2.3, 5.2.4.1, 7.1

[GFP09] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, Grenoble, France*, pages 71–80, 2009. 1.2.1, 4.1.2, 5.5, 7.2.3

[GGP08] Patrice Gerin, Xavier Guérin, and Frédéric Pétrot. Efficient Implementation of Native Software Simulation for MPSoC. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 676–681, 2008. 1.2.1, 1.2.1, 4.1.2, 4.1.2, 5.2.2

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 3.4

[GHP09] Patrice Gerin, Mian Muhammad Hamayun, and Frédéric Pétrot. Native MPSoC Co-Simulation Environment for Software Performance Estimation. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, Grenoble, France*, pages 403–412, 2009. 1.1, 1.4, 2.1, 3.4.4.2, 3.5.5, 3.6.3, 4.1.1, 4.2.2, 5.2.4.1, 5.3, 5.11, 5.3, 5.5, 7.2.3

[GKK+08] Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor Performance Estimation using Hybrid

Simulation. In *Proceedings of the 45th annual Design Automation Conference*, pages 325–330, New York, NY, USA, 2008. 4.1.3

[GKL+07] Lei Gao, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A Fast and Generic Hybrid Simulation Approach using C Virtual Machine. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 3–12, New York, NY, USA, 2007. 4.1.3, 4.1.3

[GL97] R.K. Gupta and S.Y. Liao. Using a Programming Language for Digital System Design. *IEEE Design & Test of Computers*, 14(2):72 –80, April 1997. 3.3

[Gli10] Marius Gligor. *Fast Simulation Strategies and Adaptive DVFS Algorithm for Low Power MPSoCs*. Ph.D Thesis, Institut Polytechnique de Grenoble, Grenoble, September 2010. 7.4.1

[GMH01] P. Giusto, G. Martin, and E. Harcourt. Reliable Estimation of Execution Time of Embedded Software. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 580 –588, 2001. 3.5.5, 4.2.1

[Gol74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34 –45, June 1974. 1.3, 5

[GP09] X. Guerin and F. Petrot. A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-core SoCs. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 153 –160, July 2009. 1.4, 5.2.2, 7.1

[GRE+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, 2001. 7.1.1

[GYG03] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS Modeling for System Level Design. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, page 10130, 2003. 4.1.1

[GYNJ01] P. Gerin, Sungjoo Yoo, G. Nicolescu, and A.A. Jerraya. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In *Proceedings of the 6th Asia South Pacific Design Automation Conference*, pages 63 –68, 2001. 1.2, 4.1.1

[GZD+00] Daniel Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodologie*. Kluwer Academic Publishers, Boston, March 2000. 3.3

[HAG08] Yonghyun Hwang, S. Abdi, and D. Gajski. Cycle-Approximate Retargetable Performance Estimation at the Transaction Level. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 3–8, March 2008. 4.2

[HHP+07] Kai Huang, Sang-il Han, Katalin Popovici, Lisane Brisolara, Xavier Guerin, Lei Li, Xiaolang Yan, Soo-lk Chae, Luigi Carro, and Ahmed Amine Jerraya. Simulink-based MPSoC Design Flow: Case Study of Motion-JPEG and H.264. In *Proceedings of the 44th annual Design Automation Conference*, pages 39–42, New York, NY, USA, 2007. 3.4.1

[HJ08] C. Helmstetter and V. Joloboff. SimSoC: A SystemC TLM Integrated ISS for Full System Simulation. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems*, pages 1759–1762, December 2008. 1.2.1, 4.1.2

[HSAG10] Yonghyun Hwang, Gunar Schirner, Samar Abdi, and Daniel D. Gajski. Accurate Timed RTOS Model for Transaction Level Modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 1333–1336, 2010. 3.4.4.1, 4.1.1

[HWTT04] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. RTOS-centric Hardware/Software Cosimulator for Embedded System Design. In *Proceedings of the 2nd IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 158–163, September 2004. 4.1.1

[Int11] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, October 2011. Order Number: 325384-040US. 5.1.2.1, 5.1.2.2, 5.4.2.1

[ITR] International Technology Roadmap for Semiconductors 2011 Edition: System Drivers. 1.1, 2.1

[JBP06] A.A. Jerraya, A. Bouchhima, and F. Petrot. Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In *43rd ACM/IEEE Design Automation Conference*, pages 280–285, 2006. 1.1, 2.1, 3.2, 3.4

[KAFK+05] K. Karuri, M.A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained Application Source Code Profiling for ASIP Design. In *Proceedings of the 42nd Design Automation Conference*, pages 329 – 334, June 2005. 4.2.2

[KDM90] David Ku and Giovanni De Micheli. HardwareC – A Language for Hardware Design (Version 2.0). Technical report, Stanford, CA, USA, 1990. 3.3

[KEBR08] Matthias Krause, Dominik Englert, Oliver Bringmann, and Wolfgang Rosenstiel. Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation. In *Proceedings of the 6th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 143–148, New York, NY, USA, 2008. 4.1.3

[KGW+07] Stefan Kraemer, Lei Gao, Jan Weinstock, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. HySim: A Fast Simulation Framework for Embedded Software Development. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 75–80, New York, NY, USA, 2007. 4.1.3, 4.1.3

[KKL+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007. 1.4, 5.1.2.1, 5.2, 5.2.4

[KKW+06] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A SW Performance Estimation Framework for Early System-Level-Design using Fine-Grained Instrumentation. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition, Munich, Germany*, pages 468–473, March 2006. 3.5.5, 4.1.1, 4.2, 4.2.1, 4.2.2

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, Palo Alto, California*, pages 75–87, March 2004. 5.3

[LBH+00] M.T. Lazarescu, J.R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based Software Performance Estimation for System Level Design. In *Proceedings of the IEEE International High-Level Design Validation and Test Workshop*, pages 167 –172, 2000. 4.2.3, 4.2.3.1

[LEL99] R. Leupers, J. Elste, and B. Landwehr. Generation of Interpretive and Compiled Instruction Set Simulators . In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 339–342, January 1999. 4.2.3.1

[LMPC04] R. Le Moigne, O. Pasquier, and J-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 82–87, 2004. 4.1.1, 4.2.2

[LP02] Jong-Yeol Lee and In-Cheol Park. Timed Compiled-code Simulation of Embedded Software for Performance Analysis of SOC Design. In *Proceedings of the 39th ACM/IEEE Design Automation Conference*, pages 293–298, 2002. 1.2.1, 3.5.5, 4.1.2, 4.2

[MAF91] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled Instruction Set Simulation. *Software–Practice and Experience*, 21 (8):877–889, 1991. 4.2.3.1

[MEJ+12] L.G. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers, and G. Ascheid. Synchronization for Hybrid MPSoC Full-system Simulation. In *Proceedings of the 49th ACM/IEEE Design Automation Conference*, pages 121 –126, June 2012. 4.1.3, 4.5, 4.1.3

[MEW02] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf. *Readings in Hardware/Software Co-Design*. Kluwer Academic Publishers, March 2002. 3.4

[MFP11] L. Michel, N. Fournel, and F. Petrot. Speeding-up SIMD Instructions Dynamic Binary Translation in Embedded Processor Simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition, Grenoble, France*, pages 1–4, March 2011. 4.2.3.2

[MFP12] Luc Michel, Nicolas Fournel, and Frédéric Pétrot. Fast Simulation of Systems Embedding VLIW Processors. In *Proceedings of the 10th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, Tampere, Finland*, pages 143–150, 2012. 4.2.3.2

[MME+97] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak. Simulation/Evaluation Environment for a VLIW Processor Architecture. *IBM Journal of Research and Development*, 41(3):287–302, May 1997. 4.2.3

[MMGP10] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 606–609, March 2010. 3.4.4.1

[MRRJ05] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Hybrid Simulation for Embedded Software Energy Estimation. In *Proceedings of the 42nd annual Design Automation Conference*, pages 23–26, 2005. 4.1.3

[MRRJ07a] A. Muttreja, A. Raghunathan, S. Ravi, and N.K. Jha. Hybrid Simulation for Energy Estimation of Embedded Software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(10):1843–1854, October 2007. 4.1.3

[MRRJ07b] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Automated Energy/Performance Macromodeling of Embedded Software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(3):542–552, March 2007. 4.1.3

[MSVSL08] Trevor Meyerowitz, Alberto Sangiovanni-Vincentelli, Mirko Sauermann, and Dominik Langen. Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 276–279, 2008. 3.5.5, 4.2.1

[MVG03] J. Madsen, K. Virk, and M. Gonzales. Abstract RTOS Modeling for Multiprocessor System-on-Chip. In *Proceedings of the International Symposium on System-on-Chip*, pages 147–150, November 2003. 4.1.1

[NBS+02] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proceedings of the 39th annual Design Automation Conference*, pages 22–27, New York, NY, USA, 2002. 4.7, 4.2.3.2, 4.2.3.2

[NSL+06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006. 5, 5.1

[NST06] H. Nakamura, N. Sato, and N. Tabuchi. An Efficient and Portable Scheduler for RTOS Simulation and its Certified Integration to SystemC. In *Proceedings*

of the Design, Automation and Test in Europe, volume 1, pages 1–2, March 2006. 4.1.1, 4.2.2

[NTN06] T. Nakada, T. Tsumura, and H. Nakashima. Design and Implementation of a Workload Specific Simulator. In *Proceedings of the 39th Annual Simulation Symposium*, pages 230–243, April 2006. 4.2.3.1, 6.2.3

[PAS+06] H. Posadas, J. Adamez, P. Sanchez, E. Villar, and F. Blasco. POSIX Modeling in SystemC. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 485–490, January 2006. 4.1.1

[PDV11] H. Posadas, L. Diaz, and E. Villar. Fast Data-Cache Modeling for Native Co-Simulation. In *Proceedings of the 16th Asia South Pacific Design Automation Conference*, pages 425 –430, January 2011. 4.1.2.1

[PFG+10] Frédéric Pétrot, Nicolas Fournel, Patrice Gerin, Marius Gligor, Mian Muhammad Hamayun, and Hao Shen. On MPSoC Software Execution at the Transaction Level. *IEEE Design & Test of Computers*, 28(3):2–11, 2010. 2.1, 3.9

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communication of the ACM*, 17(7):412–421, 1974. 5.1.1, A

[PGR+08] Katalin Popovici, Xavier Guerin, Frederic Rousseau, Pier Stanislao Paolucci, and Ahmed Amine Jerraya. Platform-based Software Design Flow for Heterogeneous MPSoC. *ACM Transactions on Embedded Computing Systems*, 7(4):39:1–39:23, August 2008. 3.4.2

[PHS+04] H. Posadas, F. Herrera, P. Sanchez, E. Villar, and F. Blasco. System-level Performance Analysis in SystemC. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 378–383, February 2004. 4.2.1

[PJ07] Katalin Popovici and Ahmed Amine Jerraya. Simulink based Hardware-Software Codesign Flow for Heterogeneous MPSoC. In *Proceedings of the 2007 Summer Computer Simulation Conference*, pages 497–504, San Diego, CA, USA, 2007. 1.2.1, 3.4.1, 4.1.2

[PMGP10] I.M. Pessoa, A. Mello, A. Greiner, and F. Pecheux. Parallel TLM Simulation of MPSoC on SMP Workstations: Influence of Communication Locality. In *Proceedings of the International Conference on Microelectronics*, pages 359–362, December 2010. 3.4.4.1

[PMP+04] Joshua J. Pieper, Alain Mellan, JoAnn M. Paul, Donald E. Thomas, and Faraydon Karim. High Level Cache Simulation for Heterogeneous Multiprocessors. In *Proceedings of the 41st annual Design Automation Conference*, pages 287–292, New York, NY, USA, 2004. 4.2

[Pol] PolyBench/C: The Polyhedral Benchmark Suite, `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`. 7.1.4

[PV09] Héctor Posadas and Eugenio Villar. Automatic HW/SW Interface Modeling for Scratch-Pad and Memory Mapped HW Components in Native Source-Code Co-simulation. In *Analysis, Architectures and Modelling of Embedded Systems*, volume 310, pages 12–23. 2009. 4.1.2.1, 5.2.4.1, 5.5

[PVRM10] H. Posadas, E. Villar, D. Ragot, and M. Martinez. Early Modeling of Linux-Based RTOS Platforms in a SystemC Time-Approximate Co-simulation Environment. In *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 238 –244, May 2010. 4.1.1

[RAB] RABBITS is an Annotation-Based Binary-Translation system Simulation., http://tima-sls.imag.fr/viewgit/rabbits/. 7.2

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium, Denver, Colorado, USA*, August 2000. A

[RMD03] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *Proceedings of the 40th ACM/IEEE Design Automation Conference*, pages 758–763, 2003. 4.2.3.1, 4.2.3.2

[RMD09] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation. *ACM Transactions on Embedded Computing Systems*, 8(3):20:1–20:27, April 2009. 4.2.3.2, 4.2.3.2, 4.10

[SBVR08] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-Performance Timing Simulation of Embedded Software. In *Proceedings of the 45th ACM/IEEE Design Automation Conference*, pages 290–295, June 2008. 4.2.1

[SCHY12] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: An LLVM-based Static Binary Translator. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 51–60, New York, NY, USA, 2012. 4.2.3.1, 6.1

[SD08] G. Schirner and R. Domer. Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 122–127, March 2008. 4.1.1

[SGP08] Hao Shen, Patrice Gerin, and Frédéric Pétrot. Configurable Heterogeneous MPSoC Architecture Exploration Using Abstraction Levels. In *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 51–57, Washington, DC, USA, 2008. 1.2.1, 4.1.2

[SL98] Eric Schnarr and James R. Larus. Fast Out-of-Order Processor Simulation using Memoization. *ACM SIGOPS Operating Systems Review*, 32(5):283–294, October 1998. 4.2.3.2, 6.1

[SLM06] Louis Scheffer, Luciano Lavagno, and Grant Martin. *EDA for IC System Design, Verification, and Testing*. CRC Press, March 2006. 3.4

[SYS] Open SystemC Initiative Homepage, http://www.systemc.org. 3.3, 3.3.1, 3.4

[TAB07] A. Tsikhanovich, E.M. Aboulhamid, and G. Bois. Timing Specification in Transaction Level Modeling of Hardware/Software Systems. In *Proceedings of the 50th Midwest Symposium on Circuits and Systems*, pages 249 –252, August 2007. 3.4.4.1

[TAM+08] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 51–62, 2008. 7.3.3.2

[Tan84] A. Tanenbaum. *Structured Computer Organization; (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984. 3.2

[Tex10] Texas Instruments. *Reference Guide: TMS320C64x/C64x+ DSP CPU and Instruction Set*, May 2010. SPRU732I. 3.11

[TRKA07] W. Tibboel, V. Reyes, M. Klompstra, and D. Alders. System-Level Design Flow Based on a Functional Reference for HW and SW. In *Proceedings of the 44th ACM/IEEE Design Automation Conference*, pages 23 –28, June 2007. 1.2.1, 4.1.2, 4.2

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005. 5, 5.1, 5.1.1, 5.1.2

[vdWdKH+04] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and Programming of Embedded Multiprocessors: An Interface-centric Approach. In *Proceedings of the 2nd IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 206–217, September 2004. 3.4.2

[VKS00] D. Verkest, J. Kunkel, and F. Schrirrmeister. System Level Design using C++. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 74–81, 2000. 3.3, 3.4.1

[VMW09] Performance Evaluation of Intel EPT Hardware Assist, March 2009. B.2, B.4

[WH09] Zhonglei Wang and A. Herkersdorf. An Efficient Approach for System-level Timing Simulation of Compiler-optimized Embedded Software. In *Proceedings of the 46th ACM/IEEE Design Automation Conference, San Francisco, California*, pages 220 –225, July 2009. 3.5.5, 4.1.1, 4.2.2, 4.6

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, New York, NY, USA, 1996. 4.2.3.2

[WSH08] Zhonglei Wang, Antonio Sanchez, and Andreas Herkersdorf. SciSim: A Software Performance Estimation Framework using Source Code Instrumentation. In *Proceedings of the 7th International Workshop on Software and Performance*, pages 33–42, New York, NY, USA, 2008. 1.2, 4.1.1, 4.2.1, 4.2.2

[WYW08] Peng Wang, Jianxin Yang, and Biao Wang. Simple-VLIW: A Fundamental VLIW Architectural Simulation Platform. In *Proceedings of the 7th International Asia Simulation Conference on System Simulation and Scientific Computing*, pages 1258 –1266, October 2008. 4.2.3

[YBB+03] Sungjoo Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A.A. Jerraya. Building Fast and Accurate SW Simulation Models based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 550–555, 2003. 1.2.1, 3.2, 4.1.2, 5.2.2

[YJ03] Sungjoo Yoo and A.A. Jerraya. Introduction to Hardware Abstraction Layers for SoC. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 336–337, 2003. 1.2.1, 3.5.1, 4.1.2

[YNGJ02] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 620–627, Washington, DC, USA, 2002. 4.1.1

[ZG99] Jianwen Zhu and Daniel D. Gajski. A Retargetable, Ultra-Fast Instruction Set Simulator. In *Proceedings of the Conference on Design, Automation and Test in Europe and Exhibition*, pages 298–302, New York, NY, USA, 1999. 4.2.3.1

[ZM96] V. Zivojnovic and H. Meyr. Compiled HW/SW Co-Simulation. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, pages 690–695, June 1996. 4.2.3, 4.2.3.1

[ZT00] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent Execution, No Recompilation. *IEEE Computer*, 33(3):47–52, March 2000. 4.2.3.2, 4.2.3.2, 6.1

[ZTM95] V. Zivojnovic, S. Tijang, and H. Meyr. Compiled Simulation of Programmable DSP Architectures. In *Proceedings of the Workshop on VLSI Signal Processing*, pages 187–196, oct 1995. 4.2.3.1